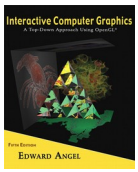


Name: _____ **Term # ____**

Homework 5 SOLUTIONS

(400 points)

NOTE: **Chapter 5 of the textbook shows the shows the viewing transformations.**
Part A is intended to be done by hand.
Part B is an openGL application.

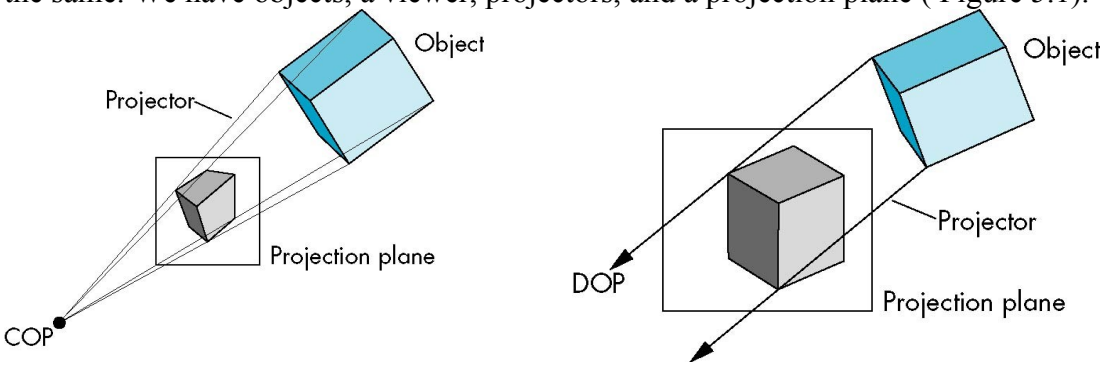
A. (300 pts) Paper and Pencil

(Guidelines: *Read the material from the textbook chapter, you can use textbook figures to exemplify your answer, use keywords, summarize your answer, but the answer* ***cannot be longer the 7 lines!****)*

5.1 CLASSICAL and COMPUTER VIEWING

a. Explain Figure 5.2.
ANSWER:
(Angel pp. 236)

When we introduced the synthetic- camera model in Chapter 1, we pointed out the similarities between classical and computer viewing. The basic elements in both cases are the same. We have objects, a viewer, projectors, and a projection plane (Figure 5.1).



The projectors meet at the center of projection (COP). The COP corresponds to the center of the lens in the camera or in the eye, and in a computer-graphics system, it is the origin of the camera frame for perspective views. All standard graphics systems follow the model that we described in Chapter 1, which is based on geometric optics. The projection surface is a plane, and the projectors are straight lines. This situation is the one we usually encounter and is straightforward to implement, especially with our pipeline model. Both classical and computer graphics allow the viewer to be an infinite distance from the objects. Note that as we move the COP to infinity, the projectors become parallel and the COP can be replaced by a direction of projection (DOP), as shown in Figure 5.2. Note also that as the COP moves to infinity, we can leave the projection plane fixed and the size of the image remains about the same, even though the COP is infinitely far from the objects. Views with a finite COP are called perspective views; views with a COP at infinity are called parallel views. For parallel views, the origin of the camera frame usually lies in the projection plane.

b. Explain "**planar geometric projection**"
ANSWER:
(Angel pp. 237)
The class of projections produced by these systems is known as planar geometric projections because the projection surface is a plane and the projectors are lines. Both perspective and parallel projections preserve lines; they do not, in general, preserve angles. Although the parallel views are the limiting case of perspective viewing, both classical and computer viewing usually treat them as separate cases. For classical views, the techniques that people use to construct the two types by hand are different, as anyone who has taken a drafting class surely knows. From the computer perspective, there are

differences in how we specify the two types of views. Rather than looking at a parallel view as the limit of the perspective view, we derive the limiting equations and use those equations directly to form the corresponding projection matrix. In modern pipeline architectures, the projection matrix corresponding to either type of view can be loaded into the pipeline.

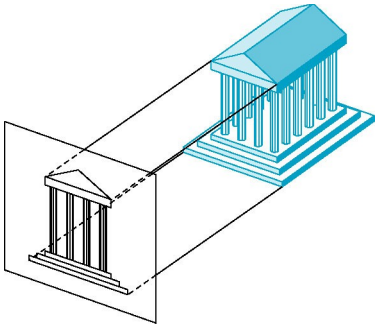
5.1.2 Orthographic Projections

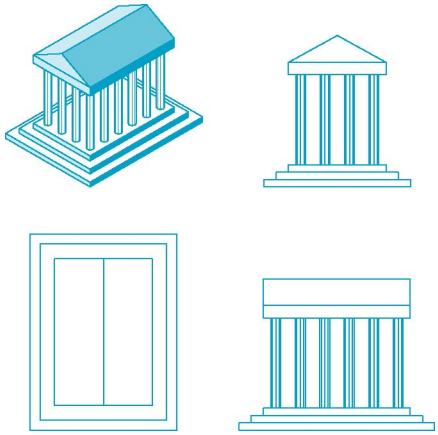
a. Define "orthographic projections"

ANSWER:

(Angel pp. 237-238)

Our first classical view is the orthographic projection shown in Figure 5.4. In all orthographic (or orthogonal) views, the projectors are perpendicular to the projection plane. In a multiview orthographic projection, we make multiple projections, in each case with the projection plane parallel to one of the principal faces of the object. Usually, we use three views such as the front, top, and right to display the object. The reason that we produce multiple views should be clear from Figure 5.5. For a box- like object, only the faces parallel to the projection plane appear in the image. A viewer usually needs more than two views to visualize what an object looks like from its multiview orthographic projections. Visualization from these images can require skill on the part of the viewer. The importance of this type of view is that it **preserves both distances and angles**, and because there is no distortion of either distance or shape, multiview orthographic projections are well suited for working drawings.





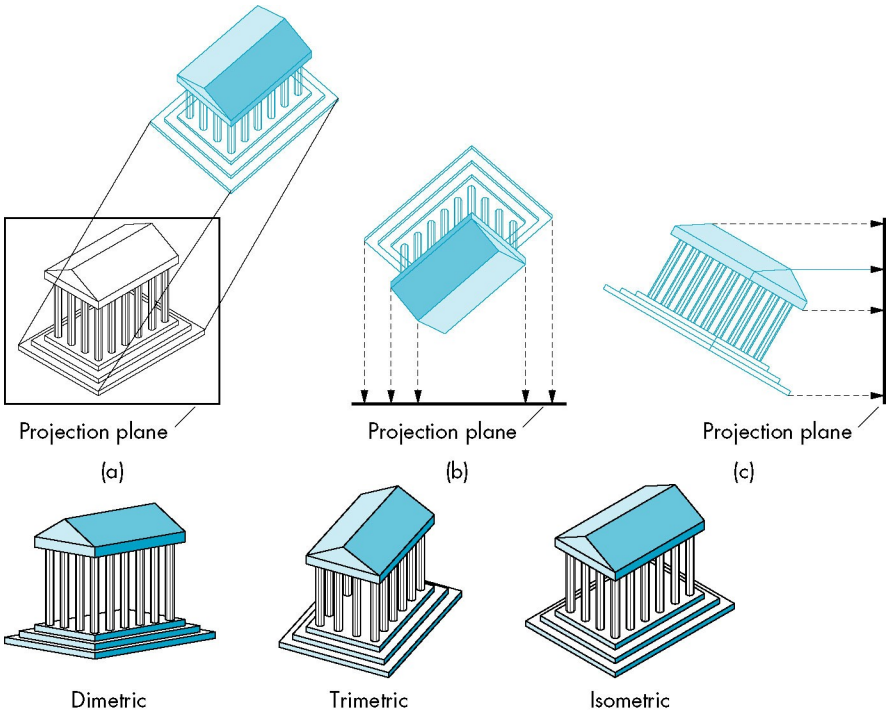
5.1.3 Axonometric Projections

a. Define "isometric view"

ANSWER:

(Angel pp. 238-239)

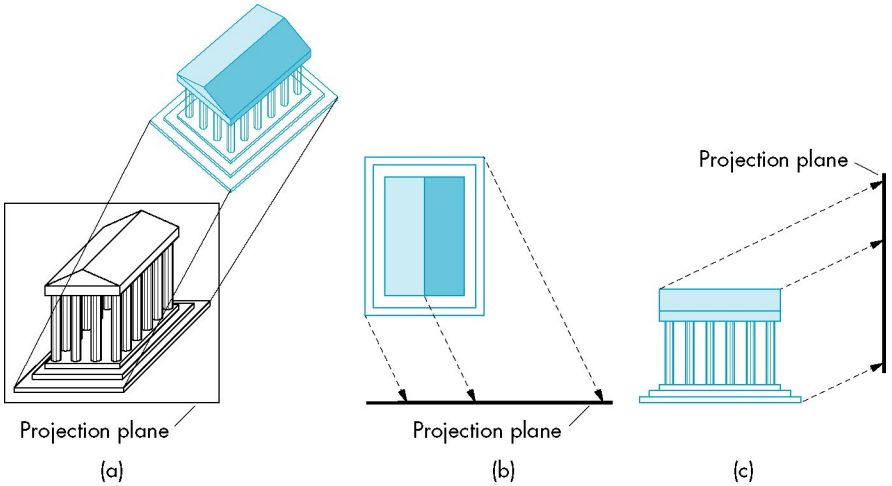
If we want to see more principal faces of our box- like object in a single view, we must remove one of our restrictions. In axonometric views, the projectors are still orthogonal to the projection plane, as shown in Figure 5.6, but the projection plane can have any orientation with respect to the object. If the projection plane is placed symmetrically with respect to the three principal faces that meet at a corner of our rectangular object, then we have an isometric view. If the projection plane is placed symmetrically with respect to two of the principal faces, then the view is dimetric. The general case is a trimetric view. These views are shown in Figure 5.7. Note that in an isometric view, a line segments length in the image space is shorter than its length measured in the object space.



b. Define "foreshortening".
ANSWER:
(Angel pp. 240)
This foreshortening of distances is the same in the three principal directions, so we can still make distance measurements. In the dimetric view, however, there are two different foreshortening ratios; in the trimetric view, there are three. Also, although parallel lines are preserved in the image, angles are not. A circle is projected into an ellipse. This distortion is the price we pay for the ability to see more than one principal face in a view that can be produced easily either by hand or by computer. Axonometric views are used extensively in architectural and mechanical design.

5.1.4 Oblique Projections
a. Define "oblique view"
ANSWER:
(Angel pp. 240)
The oblique views are the most general parallel views. We obtain an oblique projection by allowing the projectors to make an arbitrary angle with the projection plane, as shown in Figure 5.8. Consequently, angles in planes parallel to the projection plane are preserved. A circle in a plane parallel to the projection plane is projected into a circle, yet we can see more than one principal face of the object. Oblique views are the most difficult to construct by hand. They are also somewhat unnatural. Most physical viewing devices, including the human visual system, have a lens that is in a fixed relationship with the image plane usually, the lens is parallel to the plane. Although these devices produce perspective views, if the viewer is far from the object, the views are approximately parallel, but orthogonal, because the projection plane is parallel to the lens. The bellows camera that we used to develop the synthetic- camera model in Section

1.6 has the flexibility to produce approximations to parallel oblique views. One use of such a camera is to create images of buildings in which the sides of the building are parallel rather than converging as they would in an image created with a orthogonal view with the camera on the ground. From the application programmer's point of view, there is no significant difference among the different parallel views. The application programmer specifies a type of view parallel or perspective and a set of parameters that describe the camera. The problem for the application programmer is how to specify these parameters in the viewing procedures so as best to view an object or to produce a specific classical view.



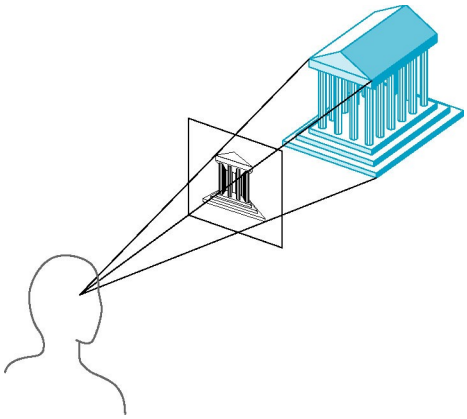
5.1.5 Perspective Viewing

a. Define "diminution of size"

ANSWER:

(Angel pp. 241)

All perspective views are characterized by diminution of size. When objects are moved farther from the viewer, their images become smaller. This size change gives perspective views their natural appearance; however, because the amount by which a line is foreshortened depends on how far the line is from the viewer, we cannot make measurements from a perspective view. Hence, the major use of perspective views is in applications such as architecture and animation, where it is important to achieve natural-looking images. In the classical perspective views, the viewer is located symmetrically with respect to the projection plane, as shown in Figure 5.9. Thus, the pyramid determined by the window in the projection plane and the center of projection is a symmetric or right pyramid. This symmetry is caused by the fixed relationship between the back (retina) and lens of the eye for human viewing, or between the back and lens of a camera for standard cameras, and by similar fixed relationships in most physical situations. Some cameras, such as the bellows camera, have movable film backs and can produce general perspective views. The model used in computer graphics includes this general case.



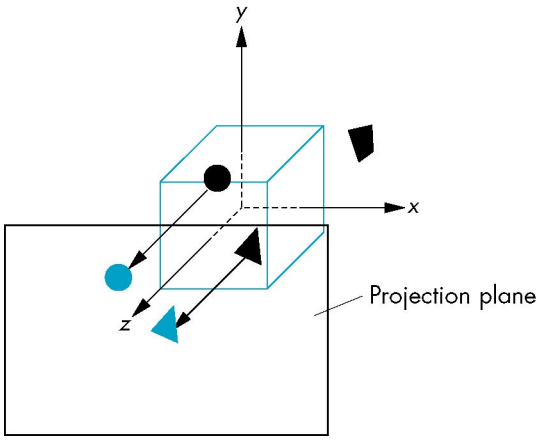
b. Define “vanishing point”.
ANSWER:
(Angel pp. 242)
The classical perspective views are usually known as one-, two-, and three- point perspectives. The differences among the three cases are based on how many of the three principal directions in the object are parallel to the projection plane. Consider the three perspective projections of the building shown in Figure 5.10. Any corner of the building includes the three principal directions. In the most general case the three- point perspective parallel lines in each of the three principal directions converges to a finite vanishing point (Figure 5.10(a)). If we allow one of the principal directions to become parallel to the projection plane, we have a two- point projection (Figure 5.10(b)), in which lines in only two of the principal directions converge. Finally, in the one- point perspective (Figure 5.10(c)), two of the principal directions are parallel to the projection plane, and we have only a **single vanishing point**. As with parallel viewing, it should be apparent from the programmer’s point of view that the three situations are merely special cases of **general perspective viewing**, which we implement in Section 5.4.

5.2 Viewing with a Computer

a. Describe the difference between computer viewing vs. classical viewing.
ANSWER:
(Angel pp. 242-243)
We can now return to three- dimensional graphics from a computer perspective. Be-cause viewing in computer graphics is based on the synthetic- camera model, we should be able to construct any of the classical views. However, there is a fundamental difference. All the classical views were based on a particular relationship among the objects, the viewer, and the projectors. In computer graphics, we stress the independence of the object specifications and camera parameters. In OpenGL, we have the choice of a perspective camera or an orthogonal camera. Whether a perspective view is a one-, two-, or three-point perspective is not something that is understood by OpenGL, as it would require knowing the relationships between objects and the camera. On balance, we prefer this independence, but if an application needs a particular type of view, the application programmer may well have to determine where to place the camera.

b. Describe the 2 steps in the pipeline architecture.
ANSWER:
(Angel pp. 243)
In terms of the pipeline architecture, viewing consists of two fundamental operations. First, we must position and orient the camera. This operation is the job of the model-view transformation. After vertices pass through this transformation, they are represented in eye or camera coordinates. The second step is the application of the projection transformation. This step applies the specified projection orthographic or perspective to the vertices and puts objects within the specified clipping volume in a normalized clipping volume. We will examine these steps in detail in the next few sections, but at this point it would help to review the default camera, that is, the camera that OpenGL uses if we do not specify any viewing functions.

c. Describe OpenGL camera defaults.
ANSWER:
(Angel pp. 243)
OpenGL starts with the camera at the origin of the object frame, pointing in the negative z- direction. This camera is set up for orthogonal views and has a viewing volume that is a cube, centered at the origin and with sides of length 2. The default projection plane is the plane $z = 0$ and the direction of the projection is along the z-axis. Thus, objects within this box are visible and projected as shown in Figure 5.11. Until now, we were able to ignore any complex viewing procedures by exploiting our knowledge of this camera. Thus, we were able to define objects in the application programs that fit inside this cube and we knew that they would be visible. In this approach, both the model- view and projection matrices were left as the default identity matrices.



d. Describe OpenGL camera placement in the model-view matrix..
ANSWER:
(Angel pp. 243)
Subsequently, we altered the model- view matrix, initially an identity matrix, by rotations and translations, so as to place the camera where we desired. The parameters that we set in glOrtho alter the projection matrix, also initially an identity matrix, so as to allow us to see objects inside an arbitrary right parallelepiped. In this chapter, we will generate a wider variety of views by using the model- view matrix to position the camera and the projection matrix to produce both orthographic and perspective views.

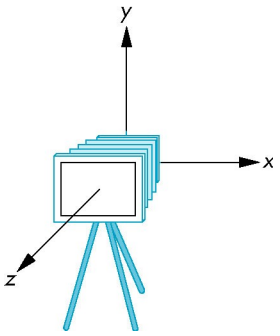
5.3 POSITIONING THE CAMERA

5.3.1 Positioning the Camera Frame

a. Describe the second part of the model-view transformation.
ANSWER:
(Angel pp. 244)
In this section, we deal with positioning and orientation of the camera; in Section 5.4, we discuss how we specify the desired projection. Although we will focus on the API that OpenGL provides for three- dimensional graphics, we also will examine briefly a few other APIs to specify a camera. In OpenGL, the model- view and projection matrices are concatenated together to form the matrix that applies to geometric entities such as vertices. We have seen one use of the model- view matrix to position objects in space. The other is to convert from the object frame to the frame of the camera.

As we saw in Chapter 4, we can specify vertices in any units we choose, and we can define a model- view matrix by a sequence of affine transformations that reposition these vertices. The model- view transformation is the concatenation of a modeling transformation that takes instances of objects in object coordinates and brings them into the world frame. The second part transforms world coordinates to eye coordi-nates. Because we usually do not need to access world coordinates, we can use the model- view matrix rather than separate modeling and viewing matrices. Initially, the model- view matrix is an identity matrix, so the camera frame and the object frame are identical. In OpenGL, the camera is initially pointing in the negative z- direction (Figure 5.12). In most applications, we model our objects as being located around the origin, so a camera

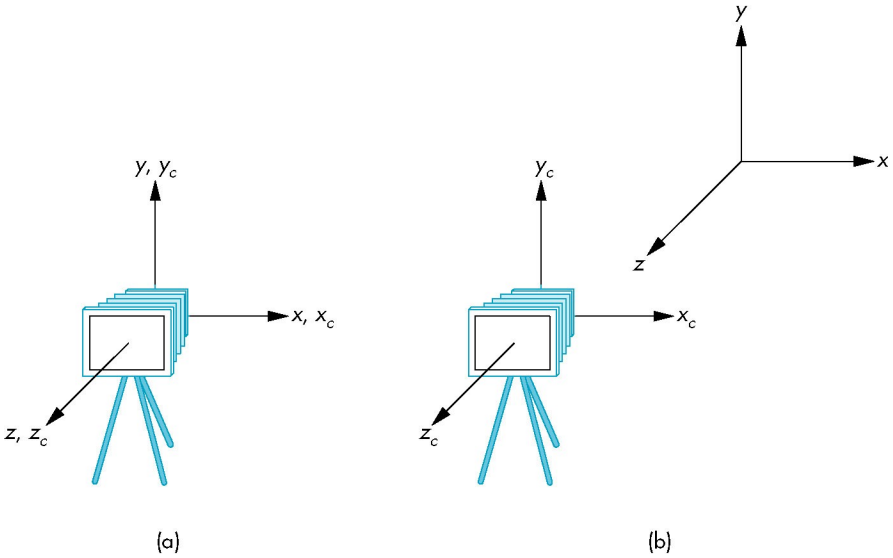
located at the default position with the default orientation does not see all the objects in the scene. Thus, either we must move the camera away from the objects that we wish to have in our image, or the objects must be moved in front of the camera. These are equivalent operations, as either can be looked at as positioning the frame of the camera with respect to the frame of the objects.



b. Describe what happens in Figure 5.13.
ANSWER:
(Angel pp. 244-245)

It might help to think of a scene in which we have initially defined several objects, with the default model- view matrix, by specifying all vertices through glVertex. Subsequent changes to the model- view matrix move the object frame relative to the camera and affect the cameras view of all objects defined afterward, because their vertices are specified relative to the repositioned object frame. Equivalently, in terms of the flow of an application program, the projection and model- view matrices are part of the state; when a primitive is defined, the system uses the current state to obtain the matrices that apply to it. Consider the sequence illustrated in Figure 5.13. In part (a), we have the initial configuration. A vertex specified at p has the same representation in both frames. In part (b), we have changed the model- view matrix to C by a sequence of transformations. The two frames are no longer the same, although C contains the information to move from the camera frame to the object frame or, equivalently, contains the information that moves the camera away from its initial position at the origin of the object frame. A vertex specified at q through glVertex, after the change to the model-view matrix, is at q in the object frame. However, its position in the camera frame is Cq and is stored internally within OpenGL; OpenGL converts positions to the camera frame through the viewing pipeline. The equivalent view is that the camera is still at the origin of its own frame, and the model- view matrix is applied to primitives specified in this system. In practice, you can use either view. But be sure to take great care regarding where in your program the primitives are specified relative to changes in the model- view matrix. At any given time, the state of the model- view matrix encapsulates the relationship between the camera frame and the object frame. Although combining the modeling and viewing transformations into a single matrix may initially cause con-

fusion, on closer examination this approach is a good one. If we regard the camera as an object with geometric properties, then transformations that alter the position and orientation of objects should also affect the position and orientation of the camera relative to these objects. The next problem is how we specify the desired position of the camera through the API and how we implement camera positioning in OpenGL. We outline three approaches, one in this section and two in Section 5.3.2. Two others are given as exercises (Exercises 5.2 and 5.3).



c. Describe what happens in Figure 5.14.

ANSWER:

(Angel pp. 246-247)

Our first approach is to specify the position indirectly by applying a sequence of rotations and translations to the model- view matrix. This approach is a direct application of the instance transformation that we presented in Chapter 4, but we must be careful for two reasons. First, we usually want to define the camera before we position any objects in the scene. 1 Second, transformations on the camera may appear to be backward from what you might expect. Consider an object centered at the origin. The camera is in its initial position, also at the origin, pointing down the negative z- axis. Suppose that we want an image of the faces of the object that point in the positive x direction. We must move the camera away from the origin. If we allow the camera to remain pointing in the negative z- direction, then we want to move the camera backward along the positive z- axis, and the proper transformation is

glTranslatef(0.0, 0.0, - d);

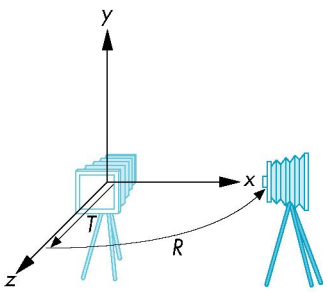
where d is a positive number.

Many people find it helpful to interpret this operation as moving the camera frame relative to the object frame. This point of view has a basis in classical viewing. In computer graphics, we usually think of objects as being positioned in a fixed frame, and

it is the viewer who must move to the right position to achieve the desired view. In classical viewing, the viewer dominates. Conceptually, we do viewing by picking up the object, orienting it as desired, and bringing it to the desired location. One consequence of the classical approach is that distances are measured from the viewer to the object, rather than as in most physically based systems from the object to the viewer. Classical viewing often resulted in a left- handed camera frame. Early graphics systems followed the classical approach by having modeling in right- handed coordinates and viewing in left-handed coordinates a decision that, although technically correct, caused confusion among users. Although in OpenGL distances from the camera, such as in glOrtho, are measured from the camera, which is consistent with classical viewing, OpenGL maintains a right-handed camera frame. Fortunately, because the application program works primarily in object coordinates, the application programmer usually does not see any of the internal representations and thus does not have to worry about these alternate perspectives on viewing. Suppose that we want to look at the same object from the positive x- axis. Now, not only do we have to move away from the object, but we also have to rotate the camera about the y- axis, as shown in Figure 5.14. We must do the translation after we rotate the camera by 90 degrees about the y- axis. In the program, the calls must be in the reverse order, as we discussed in Section 4.8, so we expect to see code like the following:

```
glMatrixMode( GL_ MODELVIEW);
glLoadIdentity();
glTranslatef( 0.0, 0.0, - d);
glRotatef(- 90.0, 0.0, 1.0, 0.0);
```

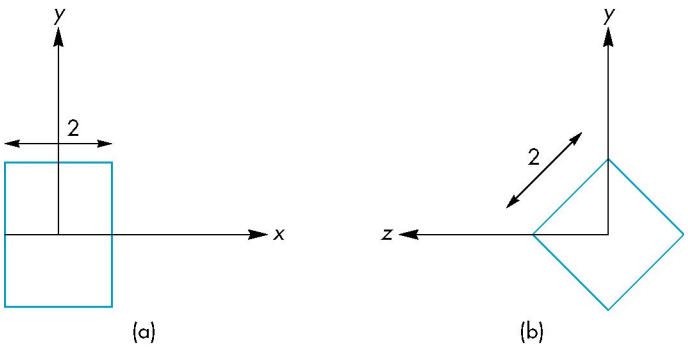
In terms of the two frames, first we rotate the object frame relative to the camera frame, and then we move the two frames apart.



d. Describe how is the isometric view of a cube created.
ANSWER:
(Angel pp. 247)
Consider creating an isometric view of the cube. Suppose that again we start with a cube centered at the origin and aligned with the axes. Because the default camera is in the middle of the cube, we want to move the camera away from the cube by a translation. We obtain an isometric view when the camera is located symmetrically with respect to three adjacent faces of the cube; for example, anywhere along the line from the origin through the point (1, 1, 1). We can move the cube away from the camera and then rotate the cube to achieve the desired view or, equivalently, move the camera away from the cube and

then rotate it to point at the cube. Starting with the default camera, suppose that we are now looking at the cube from somewhere on the positive z- axis. We can obtain one of the eight isometric views there is one for each vertex by first rotating the cube about the x- axis until we see the two faces symmetrically, as shown in Figure 5.15(a). Clearly, we obtain this view by rotating the cube by 45 degrees. The second rotation is about the y- axis. We rotate the cube until we get the desired isometric. The required angle of rotation is - 35.26 degrees about the y- axis. This second angle of rotation may not seem obvious. Consider what happens to the cube after the first rotation. From our position on the positive z- axis, the cube appears as shown in Figure 5.15(a). The original corner vertex at (- 1, 1, 1) has been transformed to (- 1, 0, v 2). If we look at the cube from the x- axis as in Figure 5.15(b), we see that we want to rotate the right vertex to the y- axis. The right triangle that determines this angle has sides of 1 and v 2, which correspond to an angle of 35.26 degrees. However, we need a clockwise rotation, so the angle must be negative. Finally, we move the camera away from the origin. Thus, our strategy is first to rotate the frame of the camera relative to the frame of the object and then to separate the two frames; the model- view matrix is of the form

$M = TR_xR_y$.



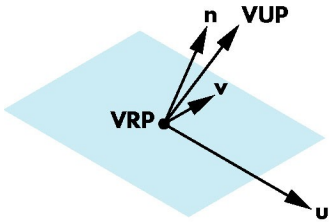
e. Describe how is the clipping volume set in OpenGL.
ANSWER:
(Angel pp. 248)
Note that the clipping volume as set by glOrtho is relative to the camera frame. Thus, for an orthographic view, the translation of the camera does not affect the size of the image of any object, but it can affect whether or not objects are clipped because the clipping volume is measured relative to the camera.

5.3.2 Two Viewing APIs

a. Define "view-reference point", **VRP**.
ANSWER:
(Angel pp. 249)
The construction of the model- view matrix for an isometric view is a little unsatisfying. Although the approach was intuitive, an interface that requires us to compute the individual angles before specifying the transformations is a poor one for an application program. We can take a different approach to positioning the camera an approach that is

similar to that used by PHIGS and GKS- 3D, two of the original standard APIs for three-dimensional graphics. Our starting point is again the object frame. We describe the cameras position and orientation in this frame. The precise type of image that we wish to obtain perspective or parallel is determined separately by the specification of the equivalent of the projection matrix in OpenGL. This second part of the viewing process is often called the normalization transformation. We approach this problem as one of a change in frames. Again, we think of the camera as positioned initially at the origin, pointed in the negative z- direction. Its desired location is centered at a point called the view- reference point (**VRP**; Figure 5.16), whose position is given in the object frame. The user executes a function such as

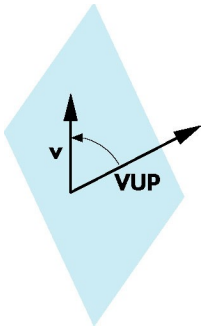
set_ view_ reference_ point(x, y, z); to specify this position.



b. Define "view-plane normal", **VPN** **&** "view-up vector", **VUP**.
ANSWER:
(Angel pp. 249-250)
Next, we want to specify the orientation of the camera. We can divide this specification into two parts: specification of the view- plane normal (VPN) and specification of the view- up vector (VUP). The VPN (n in Figure 5.16) gives the orientation of the projection plane or back of the camera. The orientation of a plane is determined by that planes normal, and thus part of the API is a function such as set_ view_ plane_ normal(nx, ny, nz); The orientation of the plane does not specify what direction is up from the cameras perspective. Given only the VPN, we can rotate the camera with its back in this plane. The specification of the VUP fixes the camera and is performed by a function such as set_ view_ up(vup_ x, vup_ y, vup_ z); We project the VUP vector on the view plane to obtain the up- direction vector v (Figure 5.17). Use of the projection allows the user to specify any vector not parallel to v, rather than being forced to compute a vector lying in the projection plane. The vector v is orthogonal to n. We can use the cross product to obtain a third orthogonal direction u. This new orthogonal coordinate system usually is referred to as either the viewing- coordinate system or the u- v- n system. With the addition of the VRP, we have the desired camera frame. The matrix that does the change of frames is the view-orientation matrix and is equivalent to the viewing component of the model- view matrix. We can derive this matrix using rotations and translations in homogeneous coordinates. We start with the specifications of the view-reference point,

We construct a new frame with the view- reference point as its origin, the view-plane normal as one coordinate direction, and two other orthogonal directions that we call u and

v. Our default is that the original x, y, z axes become u, v, n, respectively. This choice corresponds to the default model- view matrix in OpenGL. The view- reference point can be handled through a simple translation T(- x, - y, - z) from the viewing frame to the original origin. The rest of the model- view matrix is determined by a rotation so that the model- view matrix V is of the form V = TR. The direction v must be orthogonal to n; hence, n . v = 0. Figure 5.17 shows that v is the projection of vup into the plane formed by n and vup and thus must be a linear combination of these two vectors, v = an + vup. v = an + vup. If we temporarily ignore the length of the vectors, then we can set = 1 and solve for a =- vup . n n . n and v = vup - vup . n n . n n.



5.3.3 The Look-At Function

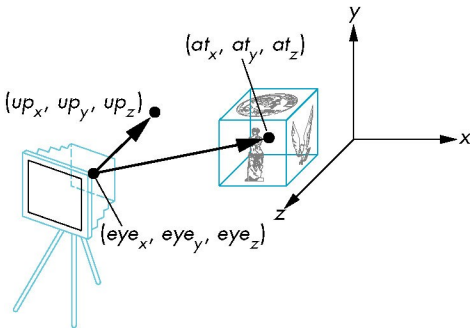
a. Describe what happens in Figure 5.18.

ANSWER:

(Angel pp. 252-253)

The use of the VRP, VPN, and VUP is but one way to provide an API for specifying the position of a camera. In many situations, a more direct method is appropriate. Consider the situation illustrated in Figure 5.18. Here a camera is located at a point e called the eye point, specified in the object frame, and it is pointed at a second point a, called the at point. These points determine a VPN and a VRP. The VPN is given by the vector formed by point subtraction between the eyepoint and the at point vpn = a - e. The view-reference point is the eye point. Hence, we need only to add the desired up direction for the camera. The OpenGL utility function void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, /* eye point */ GLdouble atx, GLdouble aty, GLdouble atx, /* at point */ GLdouble upx, GLdouble upy, GLdouble upz) /* up direction */

alters the model- view matrix for a camera pointed along this line. Thus, we usually use the sequence glMatrixMode(GL_ MODELVIEW); glLoadIdentity(); gluLookAt(eyex, eyey, eyez, atx, aty, atx, upx, upy, upz); /* define objects here */ Note that we can use the standard rotations, translations, and scalings as part of defining our objects. Although these transformations will also alter the model- view matrix, it is often helpful conceptually to consider the use of gluLookAt as posi-tioning the objects and subsequent operations that affect the model- view matrix as positioning the camera.



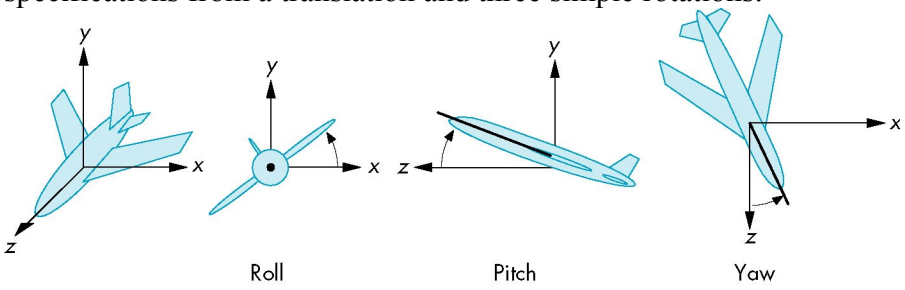
5.3.4 Other Viewing APIs

a. Describe what happens in Figure 5.19.

ANSWER:

(Angel pp. 253)

In many applications, neither of the viewing interfaces that we have presented is appropriate. Consider a flight- simulation application. The pilot using the simulator usually uses three angles roll, pitch, and yaw to specify her orientation. These angles are specified relative to the center of mass of the vehicle and to a coordinate system aligned along the axes of the vehicle, as shown in Figure 5.19. Hence, the pilot sees an object in terms of the three angles and of the distance from the object to the center of mass of her vehicle. A viewing transformation can be constructed (Exercise 5.2) from these specifications from a translation and three simple rotations.

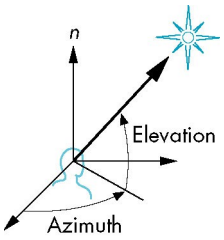


b. Describe what happens in Figure 5.20.

ANSWER:

(Angel pp. 253-254)

Viewing in many applications is most naturally specified in polar rather than rectilinear coordinates. Applications involving objects that rotate about other objects fit this category. For example, consider the specification of a star in the sky. Its direction from a viewer is given by its elevation and azimuth (Figure 5.20). The elevation is the angle above the plane of the viewer at which the star appears. By defining a normal at the point that the viewer is located and using this normal to define a plane, we define the elevation, regardless of whether or not the viewer is actually standing on a plane. We can form two other axes in this plane, creating a viewing- coordinate system. The azimuth is the angle measured from an axis in this plane to the projection onto the plane of the line between the viewer and the star. The camera can still be rotated about the direction it is pointed by a twist angle.



5.4 SIMPLE PROJECTIONS

5.4.1 Perspective Projections

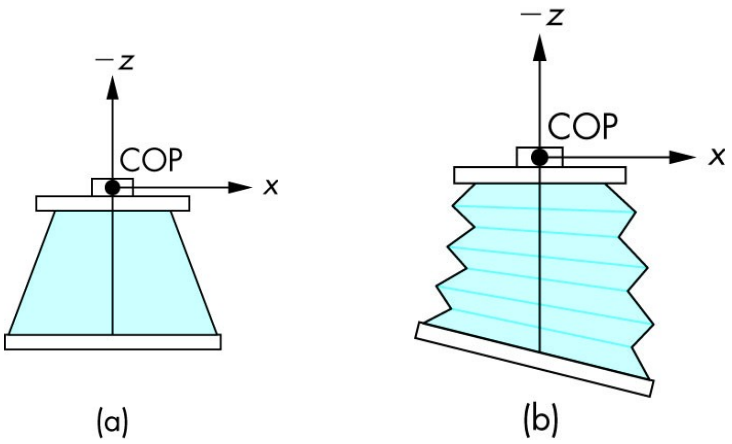
a. Describe what happens in Figure 5.21.

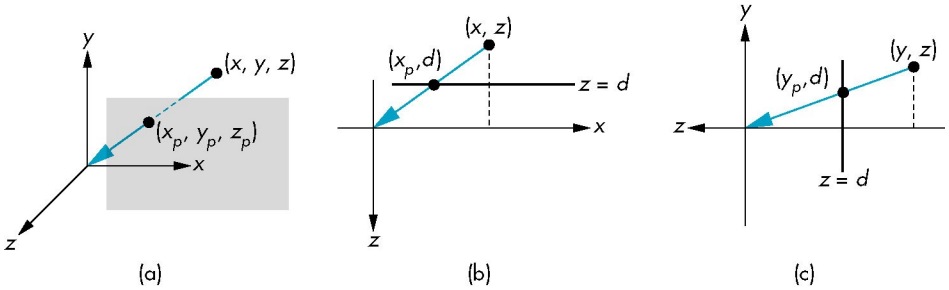
ANSWER:

(Angel pp. 254-255)

With a real camera, once we position it, we still must select a lens. As we saw in Chapter 1, it is the combination of the lens and the size of the film (or of the back of the camera) that determines how much of the world in front of a camera appears in the image. In computer graphics, we make an equivalent choice when we select the type of projection and the viewing parameters. With a physical camera, a wide- angle lens gives the most dramatic perspectives, with objects near the camera appearing large compared to objects far from the lens. A telephoto lens gives an image that appears flat and is close to a parallel view. Most APIs distinguish between parallel and perspective views by providing different functions for the two cases. OpenGL does the same, even though the implementation of the two can use the same pipeline, as we will see in Sections 5.9 and 5.10. Just as we did with the model- view matrix, we can set the projection matrix with the glLoadMatrix function. Alternatively, we can use OpenGL functions for the most common viewing conditions. First, we consider the mathematics of projection. We can extend our use of homogeneous coordinates to the projection process, which allows us to characterize a particular projection with a 4 X 4 matrix.

5.4.1 Perspective Projections Suppose that we are in the camera frame with the camera located at the origin, pointed in the negative z- direction. Figure 5.21 shows two possibilities. In Figure 5.21(a), the back of the camera is orthogonal to the z- direction and is parallel to the lens. This configuration corresponds to most physical situations, including those of the human visual system and of simple cameras. The situation shown in Figure 5.21(b) is more general; the back of the camera can have any orientation with respect to the front. We consider the first case in detail because it is simpler. How-ever, the derivation of the general result follows the same steps and should be a direct exercise (Exercise 5.6). As we saw in Chapter 1, we can place the projection plane in front of the center of projection. If we do so for the configuration of Figure 5.21(a), we get the views shown in Figure 5.22. A point in space (x, y, z) is projected along a projector into the point (xp, yp, zp). All projectors pass through the origin and, because the projection plane is perpendicular to the z- axis, $z_p = d$.





Because the camera is pointing in the negative z- direction, the projection plane is in the negative half- space z < 0, and the value of d is negative. From the top view shown in Figure 5.22(b), we see two similar triangles whose tangents must be the same. Hence, x z = xp d , and xp = x z/ d . Using the side view shown in Figure 5.22(c), we obtain a similar result for yp, yp = y z/ d . These equations are nonlinear. The division by z describes nonuniform foreshortening: The images of objects farther from the center of projection are reduced in size (diminution) compared to the images of objects closer to the COP.

b. Is the perspective transformation affine? Is it reversible?
ANSWER:
(Angel pp. 256)
We can look at the projection process as a transformation that takes points (x, y, z) to other points (xp, yp, zp). Although this perspective transformation preserves lines, it is not affine. It is also **irreversible**: Because all points along a projector project into the same point, we cannot recover a point from its projection. In Sections 5.8 and 5.9, we will develop OpenGLs use of an invertible variant of the projection transformation that preserves distances that are needed for hidden- surface removal. We can extend our use of homogeneous coordinates to handle projections. When we introduced homogeneous coordinates, we represented a point in three dimensions (x, y, z) by the point (x, y, z, 1) in four dimensions. Suppose that, instead, we replace (x, y, z) by the four- dimensional point p =..... wx wy wz w

As long as w = 0, we can recover the three- dimensional point from its four- dimensional representation by dividing the first three components by w. In this new homogeneous-

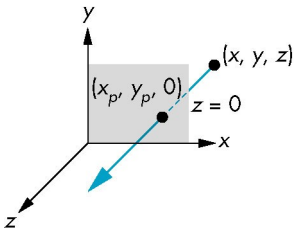
coordinate form, points in three dimensions become lines through the origin in four dimensions. Transformations are again represented by 4 4 matrices, but now the final row of the matrix can be altered and thus w can be changed by such a transformation. Obviously, we would prefer to keep w = 1 to avoid the divisions otherwise necessary to recover the three- dimensional point. However, by allowing w to change, we can represent a larger class of transformations, including perspective projections.

c. Describe what happens in Figure 5.23
ANSWER:
(Angel pp. 257)
We have shown that we can do at least a simple perspective projection, by defining a 4 X 4 projection matrix that we apply after the model- view matrix. However, we must perform a perspective division at the end. This division can be made a part of the pipeline, as shown in Figure 5.23.

5.4.2 Orthogonal Projections

a. Describe what happens in Figure 5.24
ANSWER:
(Angel pp. 257)
Orthogonal or orthographic projections are a special case of parallel projections, in which the projectors are perpendicular to the view plane. In terms of a camera, orthogonal projections correspond to a camera with a back plane parallel to the lens, which has an infinite focal length. However, rather than using limiting relations as the COP moves to infinity, we can derive the projection equations directly. Figure 5.24 shows an orthogonal projection with the projection plane z = 0. As points are projected into this plane, they retain their x and y values, and the equations of projection are
xp = x, yp = y, zp = 0. We can write this result using our original homogeneous coordinates: ..... xp yp zp 1

In this case, a division is unnecessary, although in hardware implementations, we can use the same pipeline for both perspective and orthogonal transformations. We can expand both our simple projections to general perspective and parallel projections by preceding the projection by a sequence of transformations that converts the general case to one of the two cases that we know how to apply. First, we examine the API that the application programmer uses in OpenGL to specify a projection.



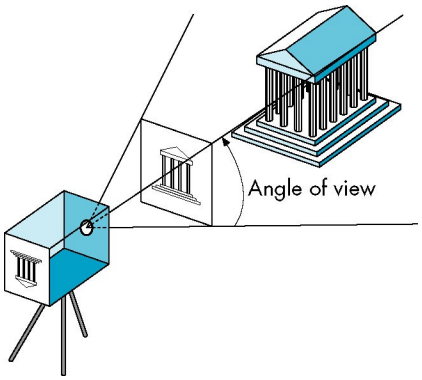
5.5 PROJECTIONS IN OPENGL

a. What is the "angle of view"? Explain Figure 5.25.

ANSWER:

(Angel pp. 258-259)

The projections that we developed in Section 5.4 did not take into account the proper-ties of the camera the - the properties of the camera the focal length of its lens or the size of the film plane. Figure 5.25 shows the angle of view for a simple pinhole camera, like the one that we discussed in Chapter 1. Only those objects that fit within the angle of view of the camera appear in the image. If the back of the camera is rectangular, only objects within a semi- infinite pyramid the view volume whose apex is at the COP can appear in the image. Objects not within the view volume are said to be clipped out of the scene. Hence, our description of simple projections has been incomplete; we did not include the effects of clipping. With most graphics APIs, the application program specifies clipping parameters through the specification of a projection.

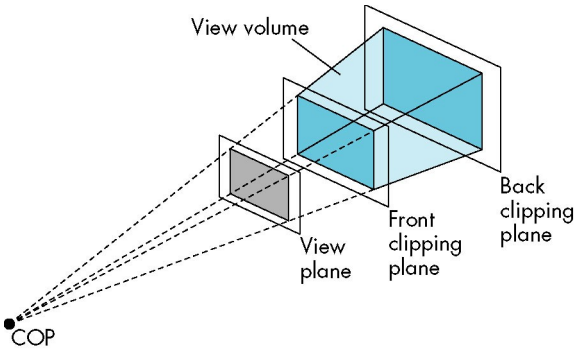


b. Explain Figure 5.26.

ANSWER:

(Angel pp. 259)

The infinite pyramid in Figure 5.25 becomes a finite clipping volume by adding front and back clipping planes, in addition to the angle of view, as shown in Figure 5.26. The resulting view volume is a frustum a truncated pyramid. We have fixed only one parameter by specifying that the COP is at the origin in the camera frame. In principle, we should be able to define each of the six sides of the frustum to have almost any orientation. If we did so, however, we would make it difficult to specify a view in the application and complicate the implementation. In practice, we rarely need this flexibility and the OpenGL API is very simple with only two perspective viewing functions. Other APIs differ in their function calls but incorporate similar restrictions. Note that whereas the OpenGL functions, such as gluLookAt, that position the camera alter the model- view matrix and are specified in object coordinates, the functions that we introduce now will alter the projection matrix. The parameters for these functions will be specified in eye coordinates.



5.5.1 Perspective in OpenGL

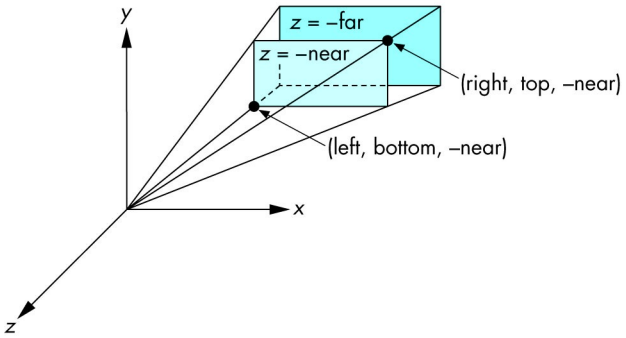
a. Explain Figure 5.27.

ANSWER:

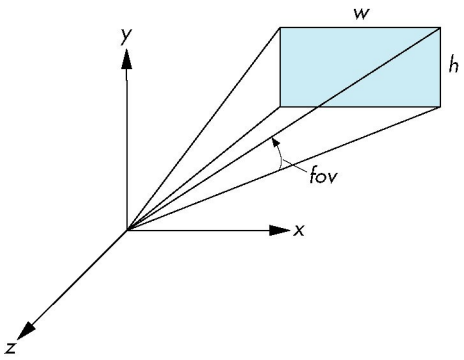
(Angel pp. 259-260)

In OpenGL, we have two functions for specifying perspective views and one for specifying parallel views. Alternatively, we can form the projection matrix directly, either by loading it, or by applying rotations, translations, and scalings to an initial identity matrix. We can specify a perspective camera view by the function glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far) These parameters are shown in Figure 5.27 in the camera frame. The near and far distances are measured from the COP (the origin in eye coordinates) to front and back clipping planes, both of which are parallel to the plane z = 0. Because the cam-era is pointing in the negative z- direction, the front (near) clipping plane is the plane z =- near, and the back (far) clipping plane is the plane z =- far. The left, right, top, and bottom values are measured in the near (front clipping) plane. The plane x = left is to the left of the camera as viewed from the COP in the direction the camera is pointing. Similar statements hold for right, bottom, and top. Although in virtually all applications far > near > 0, as long as near = far , the resulting projection matrix is valid although objects behind the center of projection the origin will be inverted in the image if they lie between the near and far planes.

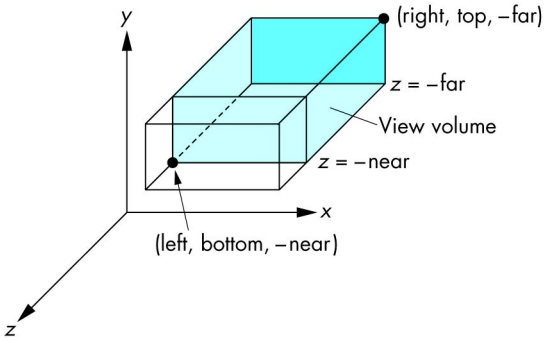
The projection matrix determined by these specifications multiplies the present matrix; thus, we must first select the matrix mode. A typical sequence follows: glMatrixMode(GL_ PROJECTION); glLoadIdentity(); glFrustum(left, right, bottom, top, near, far); Note that these specifications do not have to be symmetric with respect to the z- axis and that the resulting frustum also does not have to be symmetric (a right frustum). In Section 5.9, we show how the projection matrix for this projection can be derived from the simple perspective- projection matrix of Section 5.4.



b. Explain Figure 5.28.
ANSWER:
(Angel pp. 260-261)
In many applications, it is natural to specify the angle of view, or field of view. However, if the projection plane is rectangular, rather than square, then we see a different angle of view in the top and side views (Figure 5.28). The angle fov is the angle between the top and bottom planes of the clipping volume. The OpenGL utility function gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far) allows us to specify the angle of view in the up (y) direction, as well as the aspect ratio width divided by height of the projection plane. The near and far planes are specified as in glFrustum. This matrix also alters the present matrix, so we must again select the matrix mode, and usually must load an identity matrix, before invoking this function.



5.5.2 Parallel Viewing in OpenGL
a. Explain Figure 5.29.
ANSWER:
(Angel pp. 261)
The only parallel- viewing function provided by OpenGL is the orthogonal (ortho-graphic) viewing function glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far) Its parameters are identical to those of glFrustum. The view volume is a right parallelepiped, as shown in Figure 5.29. The near and far clipping planes are again at z =- near and z =- far, respectively.



5.6 HIDDEN-SURFACE REMOVAL

a. Explain the two broad classes of algorithms.

ANSWER:

(Angel pp. 262)

We can now return to our rotating- cube program of Section 4.10 and add perspective viewing and movement of the camera. First, we can use our development of viewing to understand the hidden- surface removal process that we used in our first version of the program. When we look at a cube that has opaque sides, we see only its three front-facing sides. From the perspective of our basic viewing model, we can say that we see only these faces because they block the projectors from reaching any other surfaces. From the perspective of computer graphics, however, all six faces of the cube have been specified and travel down the graphics pipeline; thus, the graphics sys-tem must be careful about which surfaces it displays. Conceptually, we seek algo-rithms that either remove those surfaces that should not be visible to the viewer, called hidden- surface removal algorithms, or find which surfaces are visible, called visible- surface algorithms. There are many approaches to the problem, several of which we investigate in Chapter 7. OpenGL has a particular algorithm associated with it, the z- buffer algorithm, to which we can interface through three function calls. Hence, we introduce that algorithm here, and we return to the topic in Chapter 7. Hidden- surface removal algorithms can be divided into two broad classes.

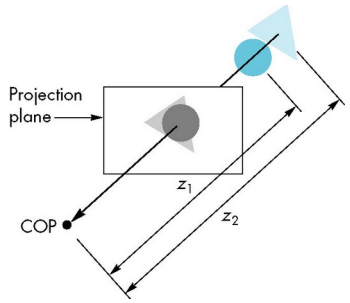
Object-space algorithms attempt to order the surfaces of the objects in the scene such that rendering surfaces in a particular order provides the correct image. For example, for our cube, if we were to render the back- facing surfaces first, we could paint over them with the front surfaces and would produce the correct image. This class of algorithms does not work well with pipeline architectures in which objects are passed down the pipeline in an arbitrary order. In order to decide on a proper order in which to render the objects, the graphics system must have all the objects available so it can sort them into the desired back- to- front order.

Image-space algorithms work as part of the projection process and seek to deter-mine the relationship among object points on each projector. The z- buffer algorithm is of the latter type and fits in well with the rendering pipeline in most graphics systems because we can save partial information as each object is rendered.

b. Explain the z-buffer algorithm and Figure 5.30.

ANSWER:

(Angel pp. 262-264)
The basic idea of the z- buffer algorithm is shown in Figure 5.30. A projector from the COP passes through two surfaces. Because the circle is closer to the viewer than to the triangle, it is the circles color that determines the color placed in the color buffer at the location corresponding to where the projector pierces the projection plane. The difficulty is determining how we can make this idea work regardless of the order in which the triangle and the circle pass through the pipeline.
Lets assume that all the objects are polygons. If, as the polygons are rasterized, we can keep track of the distance from the COP or the projection plane to the closest point on each projector that already has been rendered, then we can update this in-formation as successive polygons are projected and filled. Ultimately, we display only the closest point on each projector. The algorithm requires a depth or z buffer to store the necessary depth information as polygons are rasterized. Because we must keep depth information for each pixel in the color buffer, the z- buffer has the same spatial resolution as the color buffers. Its depth corresponds to the amount of depth resolution that is supported, usually 32 bits with recent graphics cards storing this in-formation as floating- point numbers. The z buffer is one of the buffers that constitute the frame buffer and is usually part of the memory on the graphics card. The depth buffer is initialized to a value that corresponds to the farthest distance from the viewer. When each polygon inside the clipping volume is rasterized, the depth of each fragment how far the corresponding point on the polygon is from the viewer is calculated. If this depth is greater than the value at that fragments location in the depth buffer, then a polygon that has already been rasterized is closer to the viewer along the projector corresponding to the fragment. Hence, for this fragment we ignore the color of the polygon and go on to the next fragment for this polygon, where we make the same test. If, however, the depth is less than what is already in the z- buffer, then along this projector the polygon being rendered is closer than any one we have seen so far. Thus, we use the color of the polygon to replace the color of the pixel in the color buffer and update the depth in the z- buffer. 2 For the example shown in Figure 5.30, we see that if the triangle passes through the pipeline first, its colors and depths will be placed in the color and z- buffers. When the circle passes through the pipeline, its colors and depths will replace the colors and depths of the triangle where they overlap. If the circle is rendered first, its colors and depths will be placed in the buffers. When the triangle is rendered, in the areas where there is overlap the depths of the triangle are greater than the depth of the circle, and at the corresponding pixels no changes will be made to the color or depth buffers. Major advantages of this algorithm are that its complexity is proportional to the number of fragments generated by the rasterizer and that it can be implemented with a small number of additional calculations over what we have to do to project and display polygons without hidden- surface removal. We will return to this issue in Chapter 7. From the application programmers perspective, she must initialize the depth buffer and enable hidden- surface removal by using
glutInitDisplayMode(GLUT_ DOUBLE | GLUT_ RGB | GLUT_ DEPTH);
glEnable(GL_ DEPTH_ TEST); Here we use the GLUT library for the initialization and specify a depth buffer in addition to our usual RGB color and double buffering. The programmer can clear the color and depth buffers as necessary for a new rendering by using glClear(GL_ COLOR_ BUFFER_ BIT | GL_ DEPTH_ BUFFER_ BIT);



5.6.1 Culling

a. Explain “culling”.

ANSWER:

(Angel pp. 264)

For a convex object, such as the cube, faces whose normals point away from the viewer are never visible and can be eliminated or culled before the rasterizer. We can turn on culling in OpenGL by enabling it as follows: glEnable(GL_ CULL); However, culling produces a correct image only if we have a convex object. Often we can use culling in addition to the z- buffer algorithm (which works with any collection of polygons). For example, suppose that we have a scene composed of a collection of n cubes. If we use only the z- buffer algorithm, we pass 6n polygons through the pipeline. If we enable culling, half the polygons can be eliminated early in the pipeline, and thus only 3n polygons pass through all stages of the pipeline. We consider culling further in Chapter 7.

5.7 INTERACTIVE MESH DISPLAY

5.7.1 Meshes

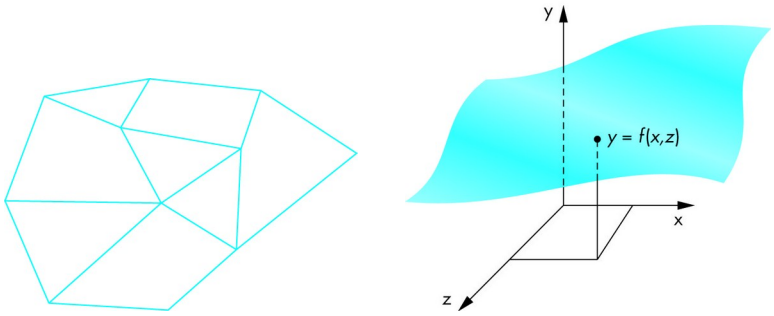
a. Explain “meshes”.

ANSWER:

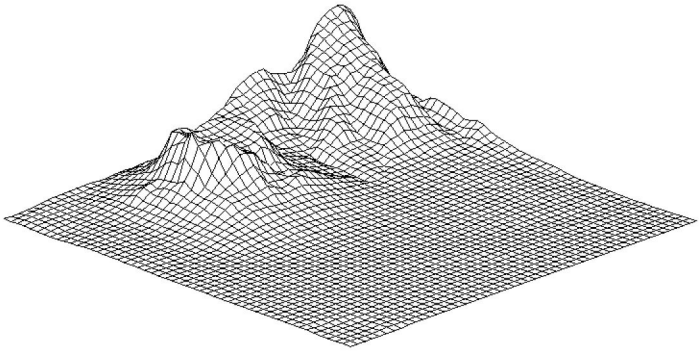
(Angel pp. 264-266)

Now we can combine our understanding of projections and modeling three- dimen-sional concepts to build an interactive application. We will use a simple mesh model that has many of the features of complex CAD models. 5.7.1 Meshes We have the tools to walk through a scene interactively by having the camera parameters change in response to user input. Before introducing the interface, lets consider another example of data display: mesh plots. A mesh is a set of polygons that share vertices and edges. A general mesh, as shown in Figure 5.31, may contain polygons with any number of vertices and require a moderately sophisticated data structure to store and display efficiently. Rectangular and triangular meshes, such as we introduced in Chapter 2 for modeling a sphere, are much simpler to work with and are useful for a wide variety of applications. Here we introduce rectangular meshes for the display of height data. Height data determine a surface, such as terrain, through either a function that gives the heights above a reference value, such as elevations above sea level, or through samples taken at various points on the surface. Suppose that the heights are given by y through a function y = f (x, z), where x and z are the points on a two- dimensional surface such as a rectangle. Thus, for each x, z we get exactly one y as shown in Figure 5.32. Such surfaces are sometimes called 2- 1/ 2 dimensional surfaces or height fields. Although all surfaces cannot be represented this

way, they have many applications. For example, if we use an x, z coordinate system to give positions on the surface of the earth, then we can use such a function to represent the height or altitude at each location. In many situations, such as when we discussed contour maps in Chapter 2, the function f is known only discretely, and we have a set of samples or measurements of experimental data of the form yij = f (xi , zj).
We assume that these data points are equally spaced such that xi = x0 + i x, i = 0, . . . , N, zj = z0 + j z , j = 0, . . . , M, where x and z are the spacing between the samples in the x and z- directions, respectively. If f is known analytically, then we can sample it to obtain a set of discrete data with which to work.



b. Explain how “meshes” are used for generating surfaces.
ANSWER:
(Angel pp. 266)
One simple way to generate a surface is through a triangular or a quadrilateral mesh. We can use the four points yij, yi+ 1, j, yi+ 1, j+ 1, and yi, j+ 1 to generate either a quadrilateral or two triangles. Thus, the data define a mesh of either NM quadrilaterals or 2NM triangles. The corresponding OpenGL programs are simple. The display callback needs only to go through the array forming alternately quads, tri-angles, quads, or quad strips from adjacent rows. For quads, the heart of the display callback is simply as follows: for(i= 0; i< N- 1; for(i=0;i<N-1; i++) for(j= 1; j< M- 1; for(j=1;j<M-1; j++) glBegin(GL_ QUADS) glVertex3i(i, y[i][j], j); glVertex3i(i,y[i][j],j); glVertex3i(i+ 1, y[i+ 1][glVertex3i(i+1,y[i+1][j], j); glVertex3i(i+ 1, y[i+ 1][glVertex3i(i+1,y[i+1][j+ 1], j+ 1); glVertex3i(i, y[i][j+ 1], glVertex3i(i,y[i][j+1], j+ 1); glEnd(); Figure 5.33 shows a rectangular mesh from height data for a part of Honolulu, Hawaii. These data are available on the Web site for the book.



5.7.1 Walking Through a Scene

a. Explain how interactivity can be added to a "scene".

ANSWER:

(Angel pp. 266-267)

The next step is to specify the camera and add interactivity. In this version, we use perspective viewing, and we allow the viewer to move the camera by pressing the x, X, y, Y, z, and Z keys on the keyboard, but we have the camera always pointing at the center of the cube. The gluLookAt function provides a simple way to reposition and reorient the camera. The changes that we have to make to our previous program (Section 4.9) are minor. We define an array viewer[3] to hold the camera position. Its contents are altered by the simple keyboard callback function keys as follows:

void keys(unsigned char key, int x, int y) { if(key == x) viewer[0]-= 1.0; if(key == X) viewer[0]+= 1.0; if(key == y) viewer[1]-= 1.0; if(key == Y) viewer[1]+= 1.0; if(key == z) viewer[2]-= 1.0; if(key == Z) viewer[2]+= 1.0; glutPostRedisplay(); }

The display function calls gluLookAt using viewer for the camera position and uses the origin for the at position. The cube is rotated, as before, based on the mouse input. Note the order of the function calls in display that alter the model- view matrix:

void display() { glClear(GL_ COLOR_ BUFFER_ BIT | GL_ DEPTH_ BUFFER_ BIT); glLoadIdentity(); gluLookAt(viewer[0], viewer[1], viewer[gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); glRotatef(theta[0], 1.0, 0.0, 0.0); glRotatef(theta[1], 0.0, 1.0, 0.0); glRotatef(theta[2], 0.0, 0.0, 1.0); /* draw mesh or other objects here */ mesh();glutSwapBuffers(); }

We can invoke glFrustum from the reshape callback to specify the camera lens through the following code:

void myReshape(int w, int h) { glViewport(0, 0, w, h); glMatrixMode(GL_ PROJECTION); glLoadIdentity(); if(w<= h) glFrustum(- 2.0, 2.0, - 2.0 * (GLfloat) h/ (GLfloat) w, 2.0* (GLfloat) h / (GLfloat) w, 2.0, 20.0); else glFrustum(- 2.0, 2.0, - 2.0 * (GLfloat) w/ (GLfloat) h, 2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0); glMatrixMode(GL_ MODELVIEW); }

Note that we chose the values of the parameters in glFrustum based on the aspect ratio of the window. Other than the added specification of a keyboard callback func-tion in main, the rest of the program is the same as the program in Section 4.10. The complete program is given in Appendix A. If you run this program, you should note the effects of moving the camera, the lens, and the sides of the viewing frustum. Note what happens as you move toward the mesh. You should also consider the effect of always having the viewer look at the center of the mesh as she is moving. Note that we could have used the mouse buttons to move the viewer. We could use the mouse buttons to move the user forward or to turn her right or left (see Exercise 5.14). However, by using the keyboard for moving the viewer, we can use the mouse to move the object as with the rotating cube in Chapter 4. In this example, we are using direct positioning of the camera through gluLookAt.

There are other possibilities. One is to use rotation and translation matrices to alter the model- view matrix incrementally. If we want to move the viewer through the scene without having her looking at a fixed point, this option may be more appealing. We could also keep a position variable in the program and change it as the viewer moves. In this case, the model- view matrix would be computed from scratch rather than changed incrementally. Which option we choose depends on the particular application, and often on other factors, such as the possibility that numerical errors might accumulate if we were to change the model- view matrix incrementally many times.

5.8 PARALLEL-PROJECTION MATRICES

5.8.1 Projection Normalization

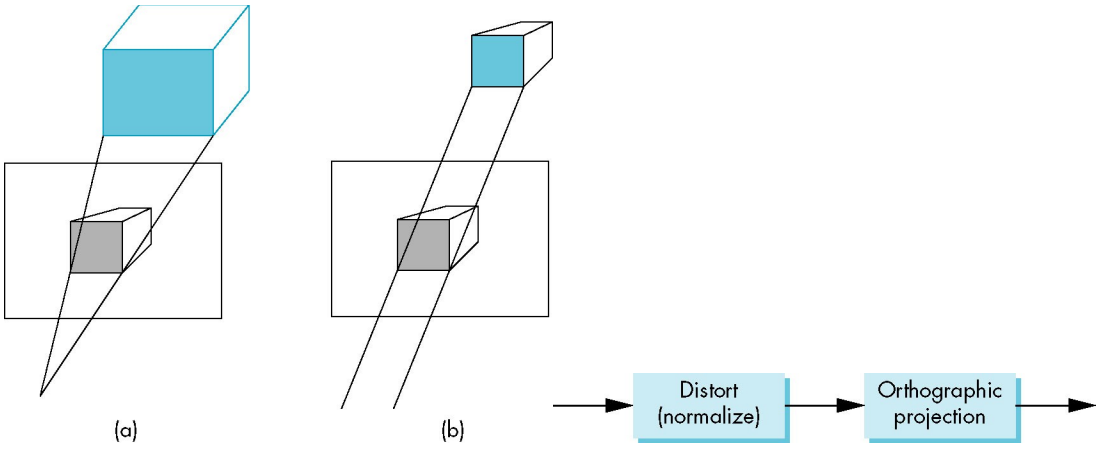
a. Explain "projection normalization".

ANSWER:

(Angel pp. 270)

5.8 PARALLEL- PROJECTION MATRICES The projection matrices we derived in Section 5.4 used very simple camera specifications that led to specific viewing volumes. The OpenGL projection matrices are not quite as simple because they support more general viewing volumes. In this section and the next, we derive the OpenGL projection matrices. Because projections are such a key part of three- dimensional computer graphics, understanding projections is crucial for both writing user applications and implementing a graphics system. Furthermore, although the OpenGL projection functions that we introduced are sufficient for most viewing situations, views such as parallel oblique are not provided directly by the OpenGL API. We can obtain such views by setting up a projection matrix from scratch or by modifying one of the standard views. We can apply either of these approaches using the matrices that we derive.

5.8.1 Projection Normalization When we introduced projection in Chapter 1 and looked at classical projection earlier in this chapter, we viewed at it as a technique that took the specification of points in three dimensions and mapped them to points on a two-dimensional projection surface. Such a transformation is not invertible, because all points along a projector map into the same point on the projection surface. In computer graphics systems, we adopt a slightly different approach. First, we work in four dimensions using homogeneous coordinates. Second, we retain depth information distance along a projector as long as possible so that we can do hidden- surface removal later in the pipeline. Third, we use a technique called projection normalization, which converts all projections into orthogonal projections by first distorting the objects such that the orthogonal projection of the distorted objects is the same as the desired projection of the original objects. This technique is shown in Figure 5.34. The concatenation of the normalization matrix, which carries out the distortion and the simple orthogonal projection matrix from Section 5.4.2, as shown in Figure 5.35, yields a homogeneous coordinate matrix that produces the desired projection.



b. Explain “Canonincal View Volume CVV”.
ANSWER:
(Angel pp. 271)
One advantage of this approach is that we can design the normalization matrix so that view volume is distorted into the canonical view volume, which is the cube defined by the planes x 1, y 1, z 1. Besides the advantage of having both perspective and parallel views supported by the same pipeline by loading in the proper normalization matrix, the canonical view volume simplifies the clipping process because the sides are aligned with the coordinate axes. The normalization process defines what most systems, including OpenGL, call the projection matrix. In OpenGL, the projection matrix brings objects into four-dimensional clip coordinates and the subsequent perspective division converts vertices to a representation in three- dimensional normalized device coordinates. Values in normalized device coordinates are later mapped to window coordinates by the viewport transformation. Here we are concerned with the first step deriving the projection matrix.

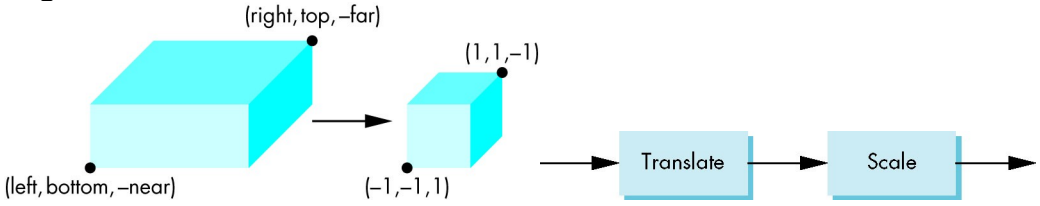
5.8.2 Orthogonal-Projection Matrices
a. What is the default “projection matrix in OpenGL”? (**CVV**)
ANSWER:
(Angel pp. 271)
5.8.2 Orthogonal- Projection Matrices Although parallel viewing is a special case of perspective viewing, we start with or-thogonal parallel viewing and later extend the normalization technique to perspective viewing. In OpenGL, the default projection matrix is an identity matrix, or equivalently, what we would get from the following code:

```
glMatrixMode( GL_ PROJECTION);
glLoadIdentity();
glOrtho(- 1.0,1.0,- 1.0, 1.0, - 1.0, 1.0);
```

The view volume is in fact the canonical view volume. Points within the cube defined by the sides x 1, y 1, and z 1 are mapped to the same cube. Points outside this cube remain outside the cube. As trivial as this observation may seem, it indicates that we can get the

desired projection matrix for the general orthogonal view by finding a matrix that maps the right parallelepiped specified by glOrtho to this same cube.

b. What is the two steps in “normalizing”?
ANSWER:
(Angel pp. 272-273)
Before we do so, recall that the last two parameters in glOrtho are distances to the near and far planes measured from a camera at the origin pointed in the negative z- direction. Thus, the near plane is at z = 1.0, which is behind the camera, and the far plane is at z =- 1.0, which is in front of the camera. Although the projectors are parallel and an orthographic projection is conceptually akin to having a camera with a long telephoto lens located far from the objects, the importance of the near and far distances in glOrtho is that they determine which objects are clipped out.
Now suppose that, instead, we set the glOrtho parameters by the following function call: glOrtho(left, right, bottom, top, near, far); We now have specified a right parallelepiped view volume whose right side (relative to the camera) is the plane x = left, whose left side is the plane x = right, whose top is the plane y = top, and whose bottom is the plane y = bottom. The front is the near clipping plane z =- near, and the back is the far clipping plane z =- far. The projection matrix that OpenGL sets up is the matrix that transforms this volume to the cube centered at the origin with sides of length 2, which is shown in Figure 5.36. This matrix converts the vertices that specify our objects, such as through calls to glVertex, to vertices within this canonical view volume, by scaling and translating them. Consequently, vertices are transformed such that vertices within the specified view volume are transformed to vertices within the canonical view volume, and vertices outside the specified view volume are transformed to vertices outside the canonical view volume. Putting everything together, we see that the projection matrix is determined by the type of view and the view volume specified in glOrtho and that these specifications are relative to the camera. The positioning and orientation of the camera are determined by the model- view matrix. These two matrices are concatenated together, and objects have their vertices transformed by this matrix product.
We can use our knowledge of affine transformations to find this projection matrix. There are two tasks that we need to do. First, we must move the center of the specified view volume to the center of the canonical view volume (the origin) by doing a translation. Second, we must scale the sides of the specified view volume to each have a length of 2 (see Figure 5.36). Hence, the two transformations are T(-(right + left)/ 2, -(top + bottom)/ 2, +(far + near)/ 2) and S(2/(right - left), 2/(top - bottom), 2/(near - far)), and This matrix maps the near clipping plane, z =- near, to the plane z =- 1 and the far clipping plane, z =- far , to the plane z = 1. Because the camera is pointing in the negative z- direction, the projectors are directed from infinity on the negative z- axis toward the origin.



$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{2}{near - far} & \frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

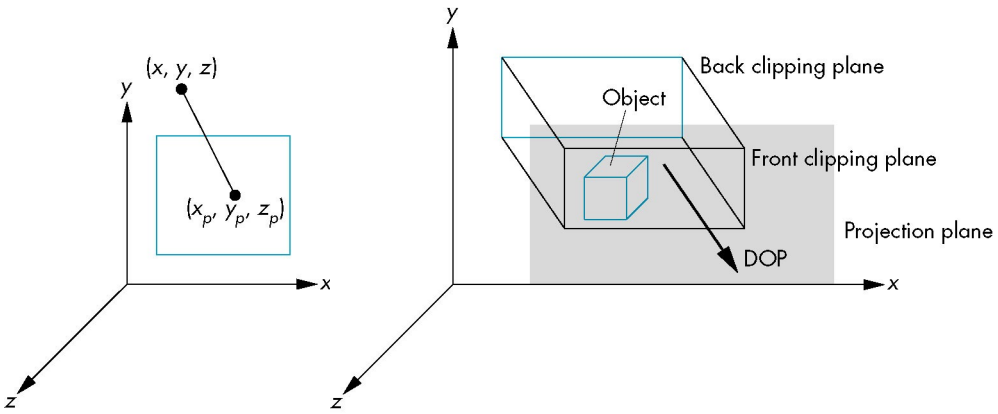
5.8.3 ObliqueProjections

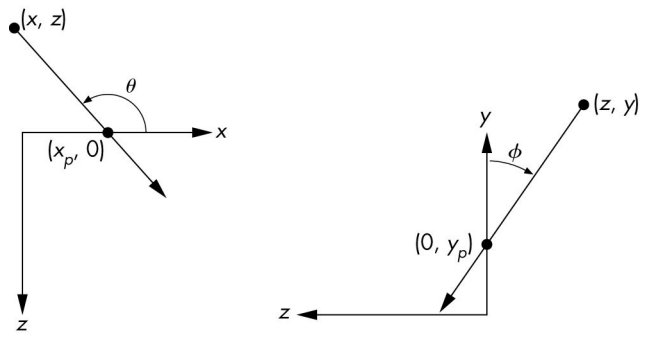
a. What transformation is also needed for normalizing?

ANSWER:

(Angel pp. 273-276)

Through glOrtho, OpenGL provides a limited class of parallel projections namely, only those for which the projectors are orthogonal to the projection plane. As we saw earlier in this chapter, oblique parallel projections are useful in many fields. 3We could develop an oblique projection matrix directly; instead, however, we follow the process that we used for the general orthogonal projection. We convert the desired projection to a canonical orthogonal projection of distorted objects. An oblique projection can be characterized by the angle that the projectors make with the projection plane, as shown in Figure 5.38. In APIs that support general parallel viewing, the view volume for an oblique projection has the near and far clipping planes parallel to the view plane, and the right, left, top, and bottom planes parallel to the direction of projection, as shown in Figure 5.39. We can derive the equations for oblique projections by considering the top and side views in Figure 5.40, which shows a projector and the projection plane z = 0. The angles . and f characterize the degree of obliqueness. In drafting, projections such as the cavalier and cabinet projections are determined by specific values of these angles. However, these angles are not the only possible interface (see Exercises 5.9 and 5.10). If we consider the top view, we can find xp by noting that tan . = z xp - x ,





(a) (b)

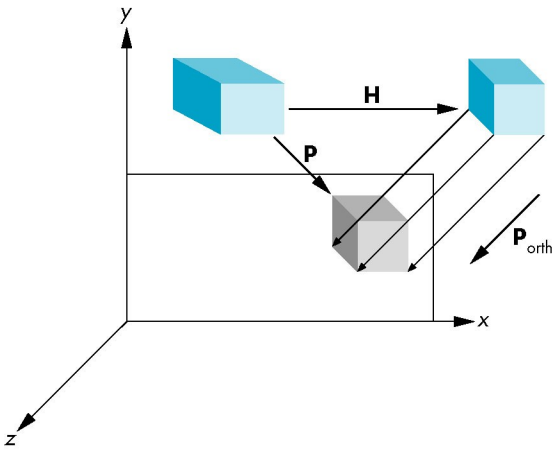
and thus xp = x + z cot . . Likewise, yp = y + z cot f. Using the equation for the projection plane zp = 0,

$$\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\ \mathbf{H}(\theta,\phi)$$

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\ \mathbf{STH}(\theta,\phi)$$

where H(. , f) is a shearing matrix. Thus, we can implement an oblique projection by first doing a shear of the objects by H(. , f) and then doing an orthographic projection. Figure 5.41 shows the effect of H(. , f) on an object a cube inside an oblique view volume. The sides of the clipping volume become orthogonal to the view plane, but the sides of the cube become oblique as they are affected by the same shear transformation. However, the orthographic projection of the distorted cube is identical to the oblique projection of the undistorted cube. We are not finished, because the view volume created by the shear is not our canonical view volume. We have to apply the same scaling and translation matrices that we used in Section 5.8.1. Hence, the transformation must be inserted after the shear and before the final orthographic projection, so the final matrix is P = MorthSTH. The values of left, right, bottom, and top are the vertices of the right parallelepiped view volume created by the shear. These values depend on how the sides of the original view volume are communicated through the application program; they may have to be determined from the results of the shear to the corners of the original view volume.



5.9 PERSPECTIVE-PROJECTION MATRICES

5.9.1 Perspective Normalization

a. Explain “perspective normalization”.

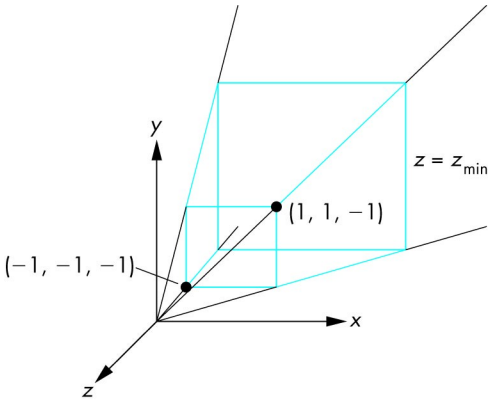
ANSWER:

(Angel pp. 276-277)

5.9 PERSPECTIVE- PROJECTION MATRICES For perspective projections, we follow a path similar to the one that we used for parallel projections: We find a transformation that allows us, by distorting the vertices of our objects, to do a simple canonical projection to obtain the desired image. Our first step is to decide what this canonical viewing volume should be. We then introduce a new transformation, the perspective-normalization transformation, that converts a perspective projection to an orthogonal projection. Finally, we derive the perspective- projection matrix used in OpenGL. 5.9.1 Perspective Normalization In Section 5.4, we introduced a simple perspective- projection matrix. For the projection plane at z =- 1 and the center of the projection at the origin, the projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

To form an image, we also need to specify a clipping volume. Suppose that we fix the angle of view at 90 degrees by making the sides of the viewing volume intersect the projection plane at a 45- degree angle. Equivalently, the view volume is the semi-infinite view pyramid formed by the planes x = +- z , y = +-z, shown in Figure 5.42. We can make the volume finite by specifying the near plane to be z =- near and the far plane to be z =- far, where both near and far, the distances from the center of projection to the near and far planes, satisfy 0 < near < far.



5.9.2 OpenGL Perspective Transformations

a. Write the OpenGL resulting projection matrix.

ANSWER:

(Angel pp. 280-281)

The OpenGL function glFrustum does not restrict the view volume to a symmetric (or right) frustum. The parameters are as shown in Figure 5.44. We can form the OpenGL perspective matrix by first converting this frustum to the symmetric frustum with 45-degree sides (see Figure 5.42). The process is similar to the conversion of an oblique parallel view to an orthogonal view. First, we do a shear to convert the asymmetric frustum to a symmetric one. Figure 5.44 shows the desired trans-formation. The shear angle is determined by our desire to skew (shear) the point ((left + right)/ 2, (top + bottom)/ 2, - near) to (0, 0, - near). The required shear matrix is H(. , f) = H cot- 1 left + right - 2near , cot- 1 top + bottom - 2near . The resulting frustum is described by the planes

$$x = \pm\frac{right - left}{-2 * near},$$

$$y = \pm\frac{top - bottom}{-2 * near},$$

$$z = -near,$$

$$z = -far.$$

The next step is to scale the sides of this frustum to

$$x = \pm z,$$

$$y = \pm z,$$

without changing either the near plane or the far plane. The required scaling matrix is $\mathbf{S}(-2 * near/(right - left), -2 * near/(top - bottom), 1)$. Note that this transformation is determined uniquely without reference to the location of the far plane $z = -far$ because in three dimensions, an affine transformation is determined by the results of the transformation on four points. In this case, these points are the four vertices where the sides of the frustum intersect the near plane.

To get the far plane to the plane $z = -1$ and the near plane to $z = 1$ after applying a projection normalization, we use the projection-normalization matrix **N**:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

with α and β as in Section 5.9.1. The resulting projection matrix is in terms of the near and far distances,

$$\mathbf{P} = \mathbf{NSH} = \begin{bmatrix} \frac{-2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{-2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & \frac{2far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

5.10 PROJECTIONS AND SHADOWS

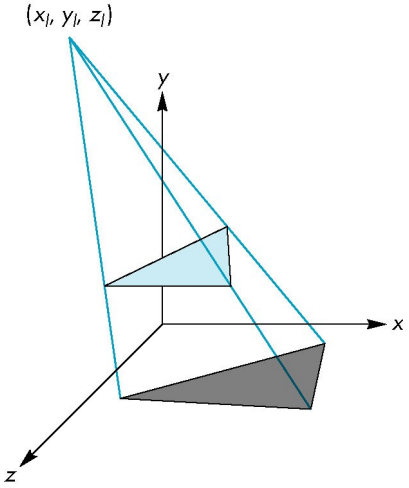
a. Explain "shadow polygon".

ANSWER:

(Angel pp. 281-282)

The creation of simple shadows is an interesting application of projection matrices.

Although shadows are not geometric objects, they are important components of realistic images and give many visual clues to the spatial relationships among the objects in a scene. Starting from a physical point of view, shadows require a light source to be present. A point is in shadow if it is not illuminated by any light source, or equivalently if a viewer at that point cannot see any light sources. However, if the only light source is at the center of projection, there are no visible shadows, because any shadows are behind visible objects. This lighting strategy has been called the flashlight in the eye model and corresponds to the simple lighting we have used thus far. To add physically correct shadows, we must understand the interaction between light and materials, a topic that we investigate in Chapter 6. There we show that global calculations are difficult; normally, they cannot be done in real time objects in a scene. Starting from a physical point of view, shadows require a light source to be present. A point is in shadow if it is not illuminated by any light source, or equivalently if a viewer at that point cannot see any light sources. However, if the only light source is at the center of projection, there are no visible shadows, because any shadows are behind visible objects. This lighting strategy has been called the flashlight in the eye model and corresponds to the simple lighting we have used thus far. To add physically correct shadows, we must understand the interaction between light and materials, a topic that we investigate in Chapter 6. There we show that global calculations are difficult; normally, they cannot be done in real time.

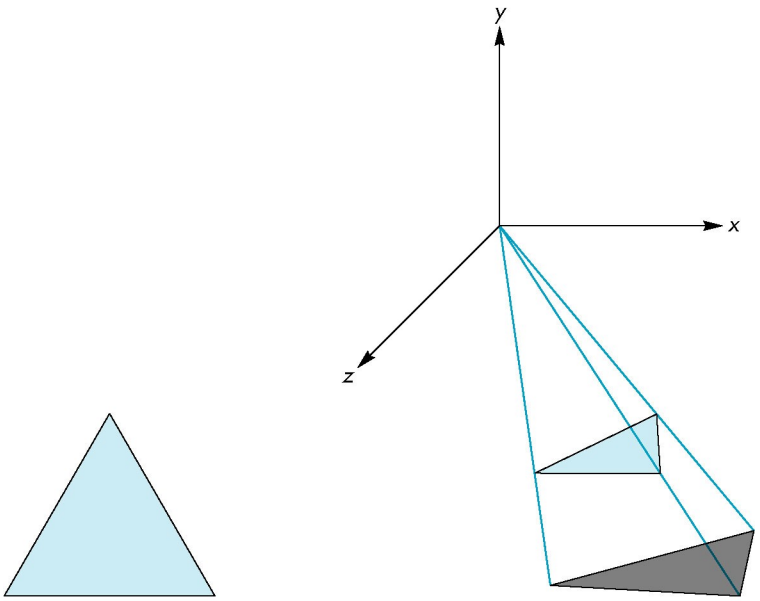


Nevertheless, the importance of shadows in applications such as flight simulators led to a number of special approaches that can be used in many circumstances. Consider the shadow generated by the point source in Figure 5.45. We assume for simplicity that the shadow falls on the ground or the surface, y = 0. Not only is the shadow a flat polygon, called a shadow polygon, but it also is the projection of the original polygon onto the surface. Specifically, the shadow polygon is the projection of the polygon onto the surface with the center of projection at the light source. Thus, if we do a projection onto the plane of a surface in a frame in which the light source is at the origin, we obtain the vertices of the shadow polygon. These vertices must then be converted back to a representation in the object frame. Rather than do the work as part of an application program, we can find a suitable projection matrix and use OpenGL to compute the vertices of the shadow polygon.

b. Explain Figure 5.46.
ANSWER:
(Angel pp. 283)

Suppose that we start with a light source at (xl , yl , zl), as shown in Figure 5.46(a). If we reorient the figure such that the light source is at the origin, as shown in Figure 5.46(b), by a translation matrix T(- xl , - yl , - zl), then we have a simple perspective projection through the origin. The projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$



(a) (b)

Finally, we translate everything back with T(xl , yl , zl). The concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l},$$

$$y_p = 0,$$

$$z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}.$$

However, with an OpenGL program, we can alter the model- view matrix to form the desired polygon as follows:

```
GLfloat m[16];                  /* shadow projection matrix */
for(i=0;i<15;i++) m[i]=0.0;
m[0]=m[5]=m[10]=1.0;
m[7]= -1.0/yl;

glColor3fv(polygon_color)
glBegin(GL_POLYGON)
.                               /* draw the polygon normally */
.
glEnd();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();                 /* save state */
glTranslatef(xl,yl,zl);         /* translate back */
glMultMatrixf(m);               /* project */
glTranslate(-xl,-yl,-zl);       /* move light to origin */
glColor3fv(shadow_color);
glBegin(GL_POLYGON);
.                               /* draw the polygon again */
.
glEnd();
glPopMatrix();                  /* restore state */
```

Note that although we are performing a projection with respect to the light source, the matrix that we use is the model- view matrix. We render the same polygon twice: the first time as usual and the second time with an altered model- view matrix that transforms the vertices. The same viewing conditions are applied to both the polygon and its shadow polygon. The results of computing shadows for the cube are shown in Back Plate 3. The code is in the program cubevs. c on the companion Web site. For a simple environment, such as an airplane flying over flat terrain casting a single shadow, this technique works well. It is also easy to convert from point sources to distant (parallel) light sources (see Exercise 5.17). However, when objects can cast shadows on other objects, this method becomes impractical. In Chapter 13, we address more general, but slower, rendering process.

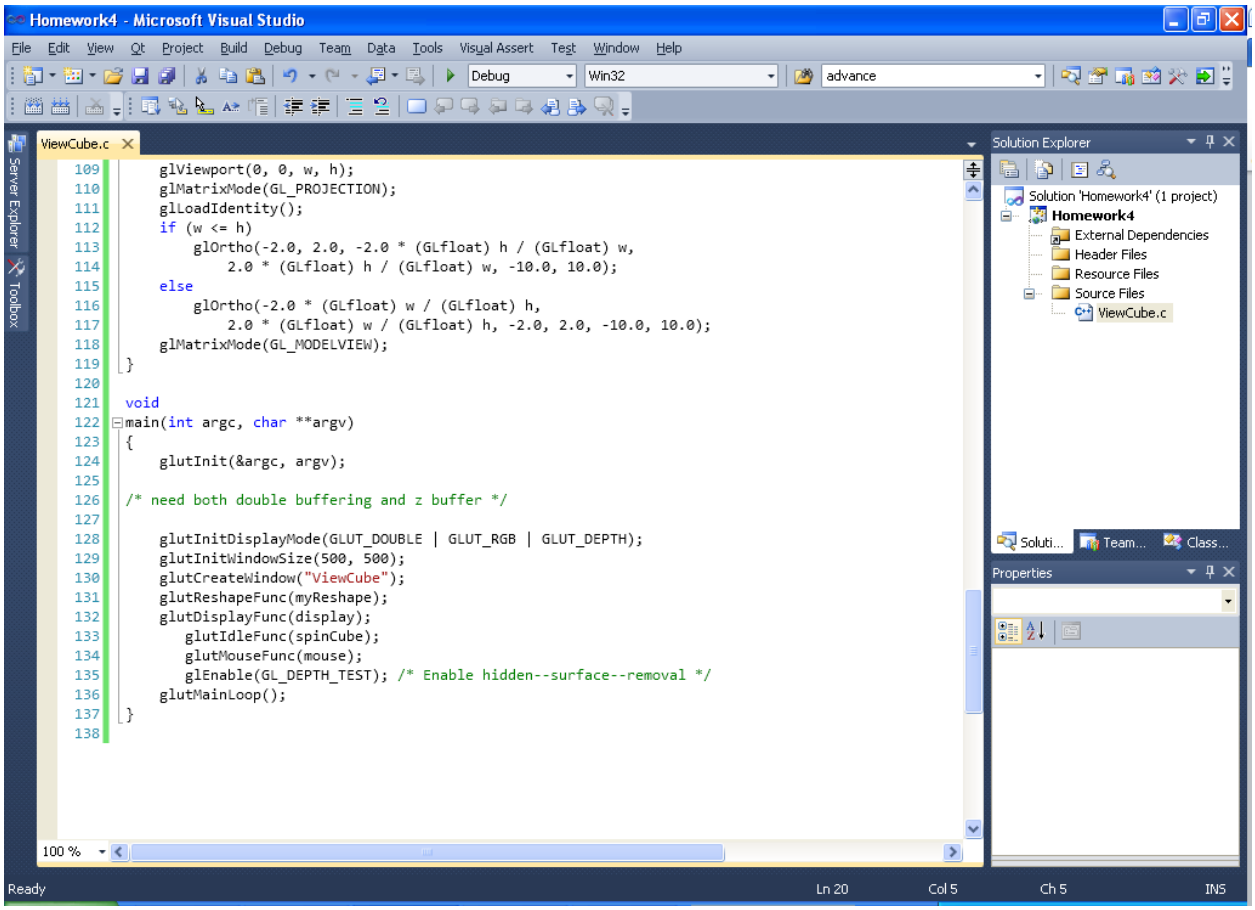
B. (50 pts) Visual Studio 2008 C++ Project

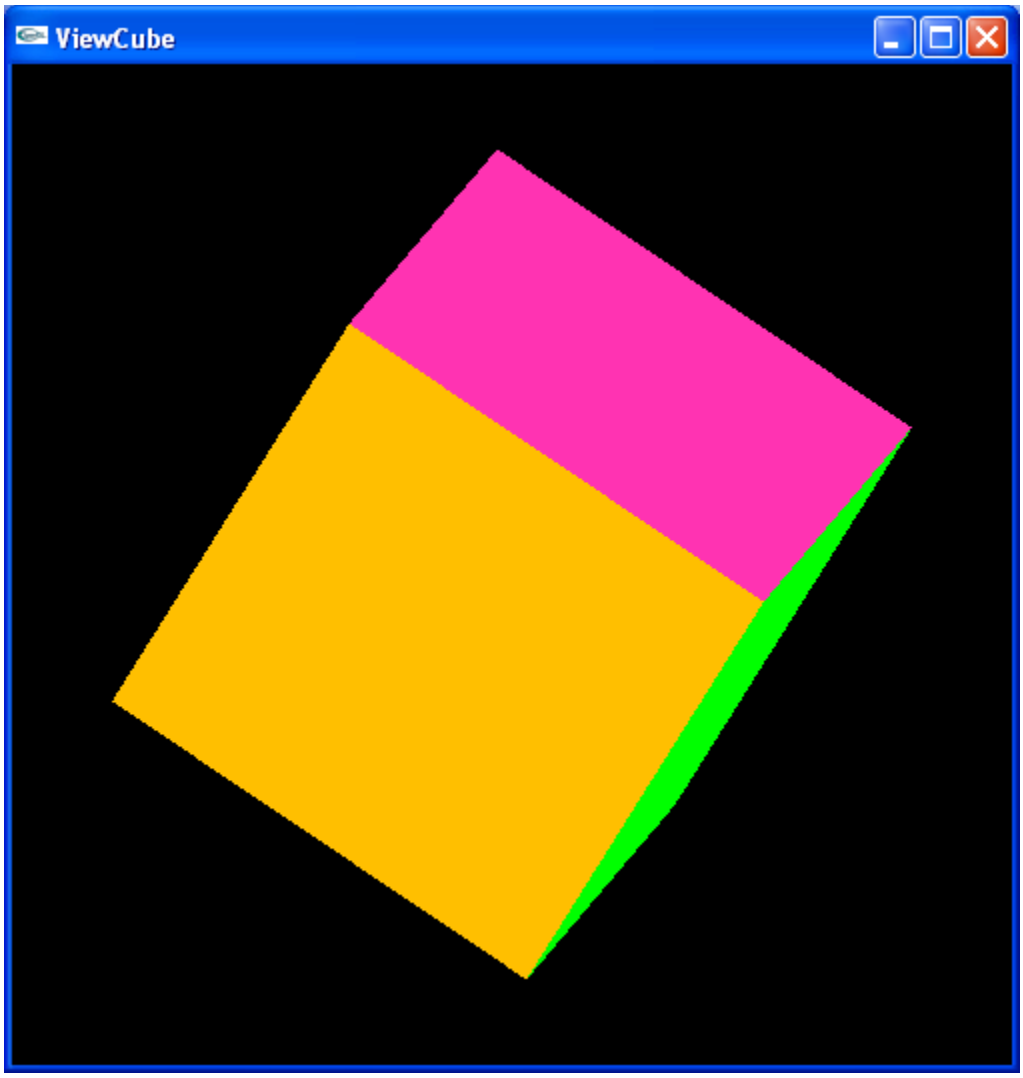
B1. Create Visual Studio 2008 C++, Empty Project, Homework5:
Create a .c file **ViewCube.c** by modifying the file below:

```
/* ViewCube */

#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>

#endif

      GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
      {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
      {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

      GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
      {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
      {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};


void polygon(int a, int b, int c , int d)
{
      glBegin(GL_POLYGON);

            glNormal3fv(normals[a]);
            glVertex3fv(vertices[a]);

            glNormal3fv(normals[b]);
            glVertex3fv(vertices[b]);

            glNormal3fv(normals[c]);
            glVertex3fv(vertices[c]);

            glNormal3fv(normals[d]);
            glVertex3fv(vertices[d]);
      glEnd();


                        }

void colorcube()
{
      /*  Front of MYcube counter clockwise front facing face*/
    glColor3f(1.0,0.0,0.0);       //red
    polygon(0,3,2,1);

    /*  Back of MYcube  clockwise back facing face*/
      glColor3f(0.0,1.0,0.0);      //green
      polygon(4,5,6,7);

    /*  Bottom of MYcube clockwise back facing face*/
```

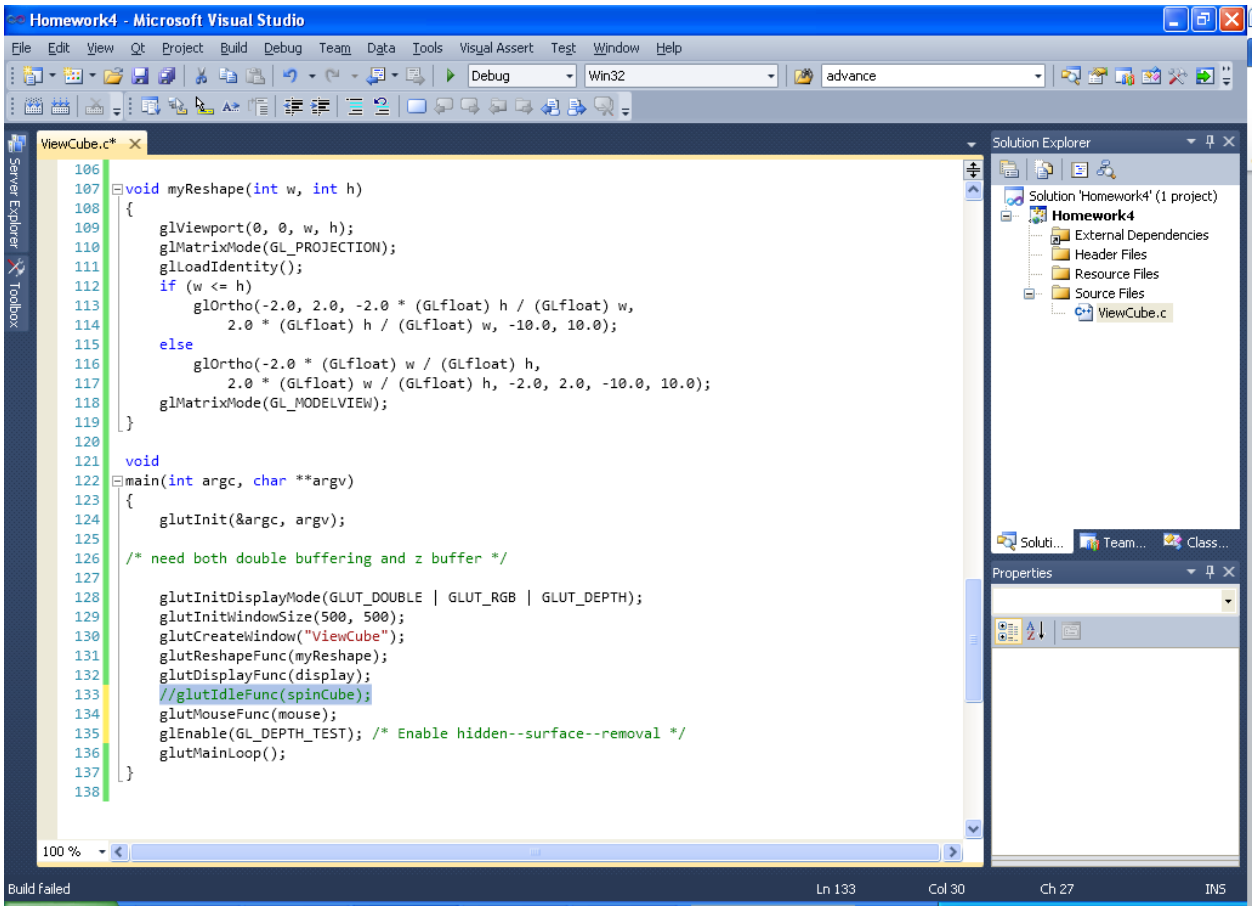
```
      glColor3f(1.0,0.2,0.7);      //deep pink
      polygon(3,0,4,7);

      /*  Top of MYcube counter clockwise front facing face*/
      glColor3f(0.0,0.75388,1.0);      //blue
    polygon(1,2,6,5);

      /*  Right Side of MYcube counter clockwise front facing face*/
      glColor3f(1.0,1.0,0.0);      // yellow
    polygon(2,3,7,6);

      /*  Left Side of MYcube clockwise back facing face*/
      glColor3f(1.0,0.75,0.0);      // orange
      polygon(5,4,0,1);
}

static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;

void display(void)
{
/* display callback, clear frame buffer and z buffer,
   rotate cube and draw, swap buffers */

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      glRotatef(theta[0], 1.0, 0.0, 0.0);
      glRotatef(theta[1], 0.0, 1.0, 0.0);
      glRotatef(theta[2], 0.0, 0.0, 1.0);

 colorcube();

 glFlush();
      glutSwapBuffers();
}

void spinCube()
{

/* Idle callback, spin cube 2 degrees about selected axis */

      theta[axis] += 2.0;
      if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
      /* display(); */
      glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

      if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
      if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
      if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}

void myReshape(int w, int h)
```

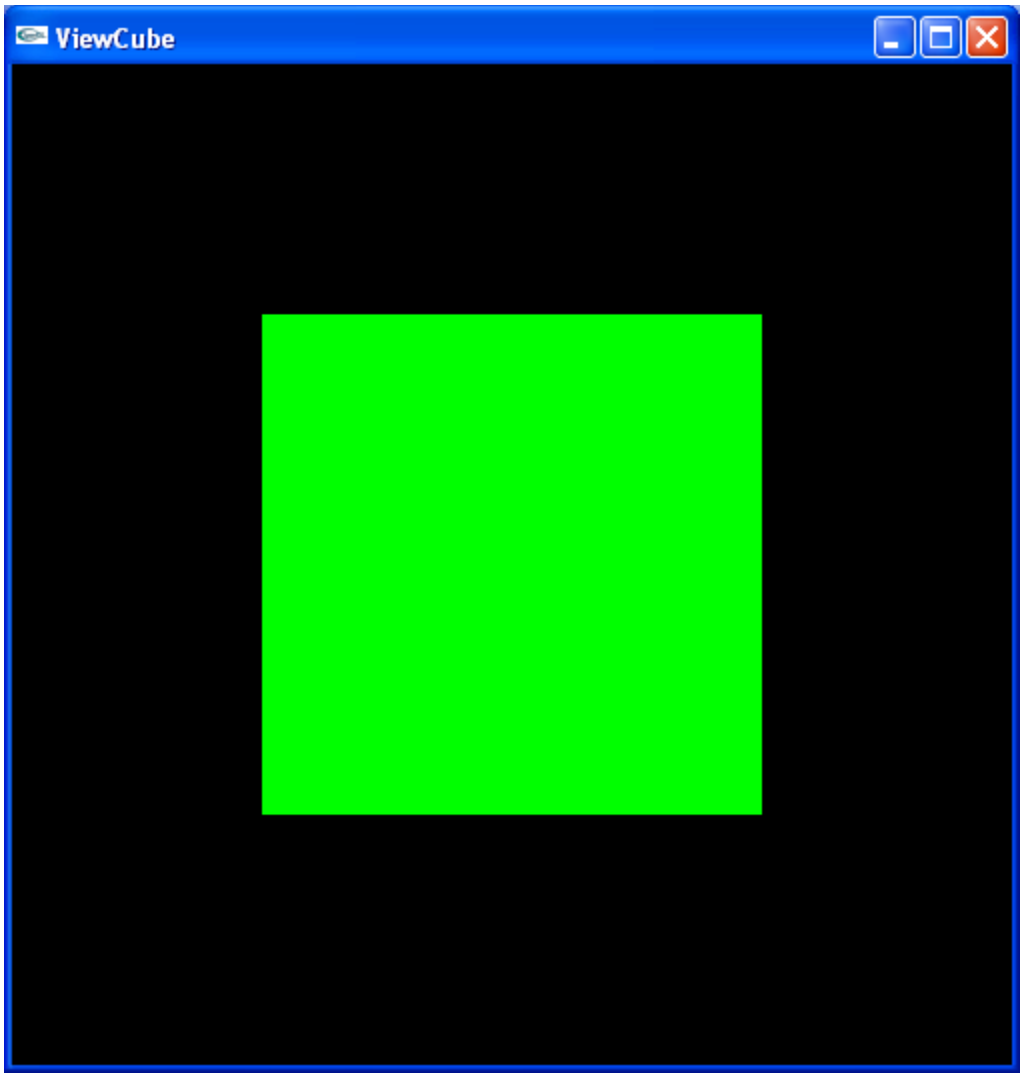
```
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

void
main(int argc, char **argv)
{
    glutInit(&argc, argv);

/* need both double buffering and z buffer */

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("ViewCube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
         glutIdleFunc(spinCube);
         glutMouseFunc(mouse);
         glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
    glutMainLoop();
}
```


Homework4 - Microsoft Visual Studio
File Edit View Qt Project Build Debug Team Data Tools Visual Assert Test Window Help
Debug
Win32
advance
ViewCube.c
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
else
glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
}
void
main(int argc, char **argv)
{
glutInit(&argc, argv);
/* need both double buffering and z buffer */
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(500, 500);
glutCreateWindow("ViewCube");
glutReshapeFunc(myReshape);
glutDisplayFunc(display);
glutIdleFunc(spinCube);
glutMouseFunc(mouse);
glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
glutMainLoop();
}
100 %
Solution Explorer
Solution 'Homework4' (1 project)
Homework4
External Dependencies
Header Files
Resource Files
Source Files
ViewCube.c
Properties
Ready
Ln 20
Col 5
Ch 5
INS



1. Remove **spinCube** (…) function


Homework4 - Microsoft Visual Studio
File Edit View Qt Project Build Debug Team Data Tools Visual Assert Test Window Help
Debug
Win32
advance
ViewCube.c*
106
107 void myReshape(int w, int h)
108 {
109 glViewport(0, 0, w, h);
110 glMatrixMode(GL_PROJECTION);
111 glLoadIdentity();
112 if (w <= h)
113 glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
114 2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
115 else
116 glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
117 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
118 glMatrixMode(GL_MODELVIEW);
119 }
120
121 void
122 main(int argc, char **argv)
123 {
124 glutInit(&argc, argv);
125
126 /* need both double buffering and z buffer */
127
128 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
129 glutInitWindowSize(500, 500);
130 glutCreateWindow("ViewCube");
131 glutReshapeFunc(myReshape);
132 glutDisplayFunc(display);
133 //glutIdleFunc(spinCube);
134 glutMouseFunc(mouse);
135 glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
136 glutMainLoop();
137 }
138
100 %
Solution Explorer
Solution 'Homework4' (1 project)
Homework4
External Dependencies
Header Files
Resource Files
Source Files
ViewCube.c
Soluti...
Team...
Class...
Properties
Server Explorer
Toolbox
Build failed
Ln 133
Col 30
Ch 27
INS



2. Initially your viewer is set at:

static GLdouble viewer[]= {0.0, 0.0, 5.0}; /* initial viewer location */

3. Add a function keys that returns void with 3 parameters as described below

void keys(unsigned char key, int x, int y)

that allows the use x, X, y, Y, z, and Z keys to move viewer. Lower case will move the viewer by **-1**, upper case by **1**.

4. In display (….) function use the gluLookAt with the following parameters:

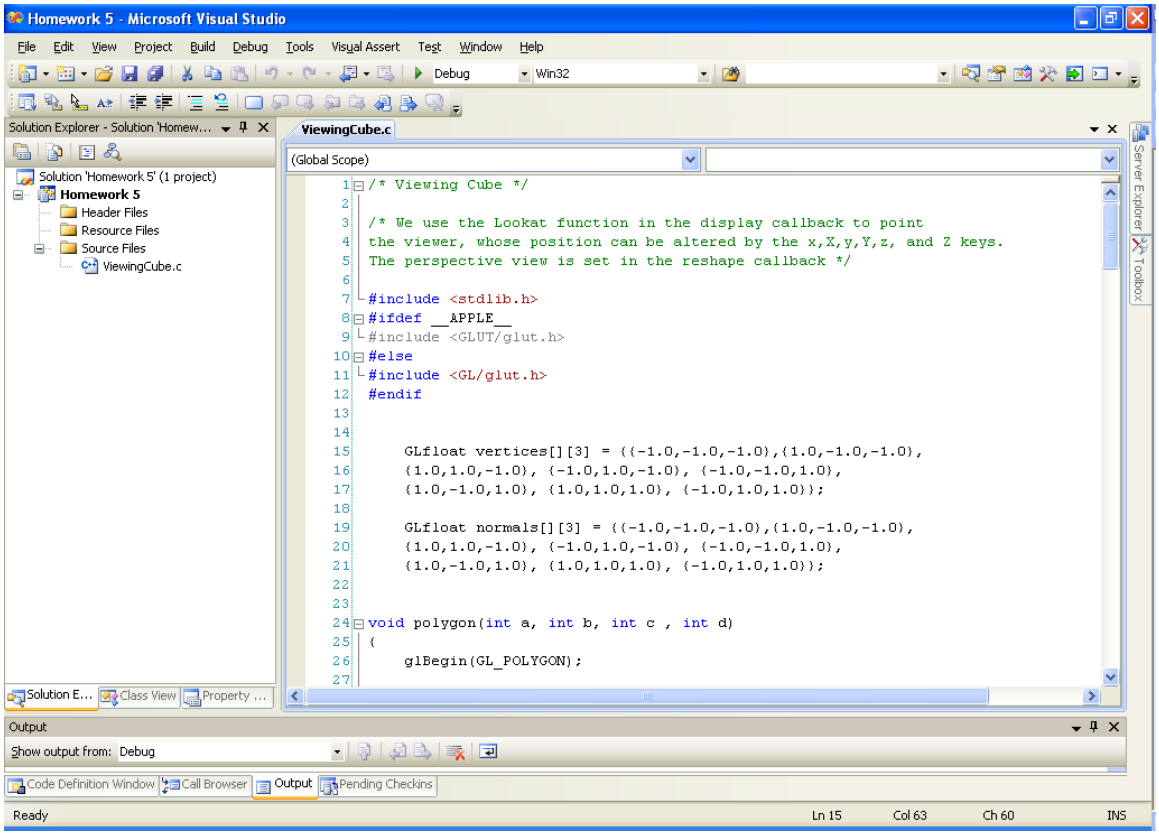
gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

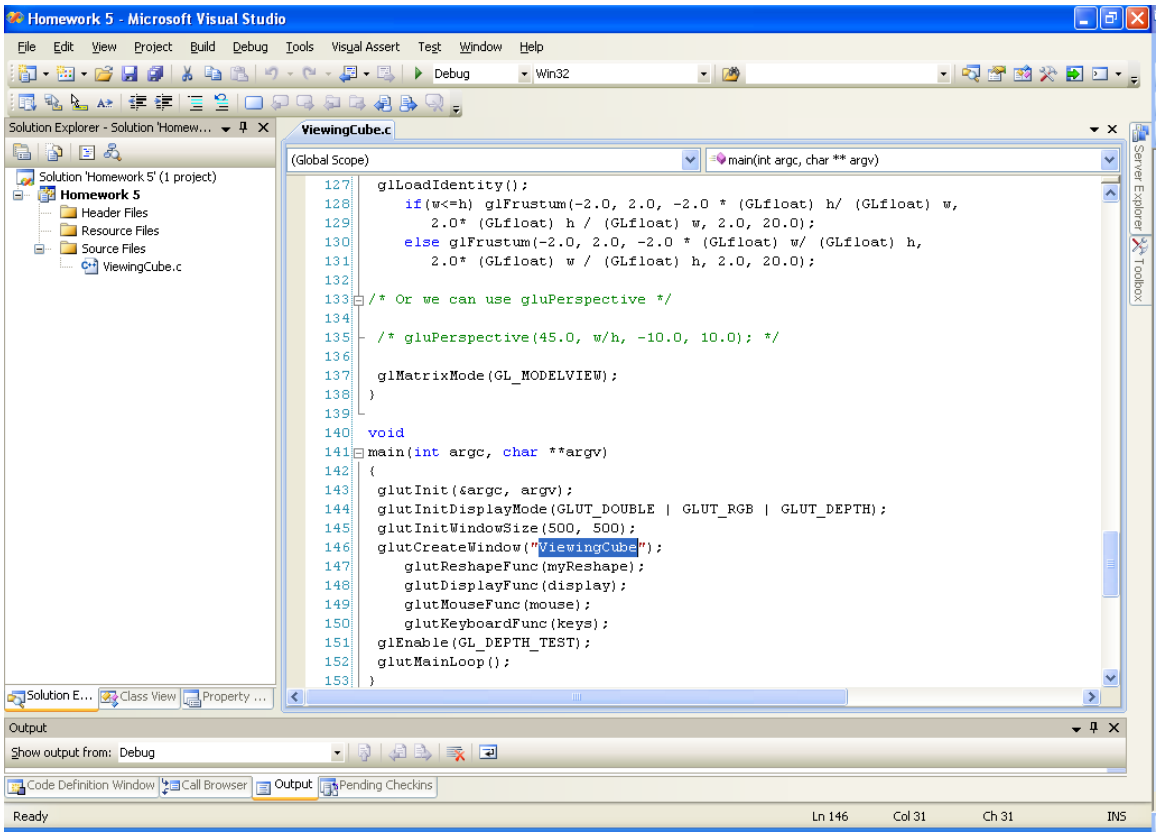
5. In myReshape (…) function use the glFrustum or gluPerspective instead of glOrtho.

Build and run this Project: Insert a screenshot of your output.

ANSWER:

File Name: ViewingCube.c



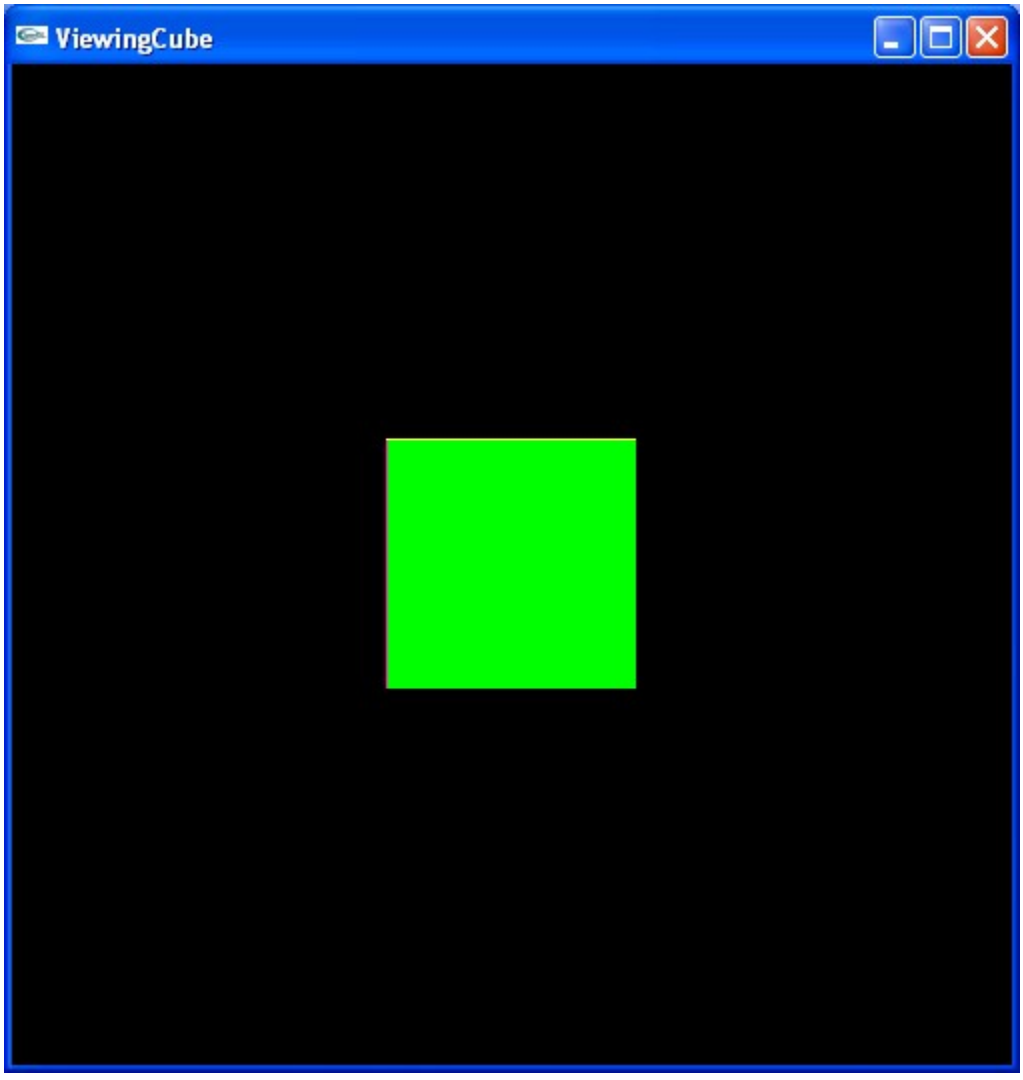



```
/* ViewingCube */

/* We use the Lookat function in the display callback to point
the viewer, whose position can be altered by the x,X,y,Y,z, and Z keys.
The perspective view is set in the reshape callback */

#include <stdlib.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif


      GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
      {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
      {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

      GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
      {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
      {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

      void polygon(int a, int b, int c , int d)
{
```

```
      glBegin(GL_POLYGON);

            glNormal3fv(normals[a]);
            glVertex3fv(vertices[a]);

            glNormal3fv(normals[b]);
            glVertex3fv(vertices[b]);

            glNormal3fv(normals[c]);
            glVertex3fv(vertices[c]);

            glNormal3fv(normals[d]);
            glVertex3fv(vertices[d]);
      glEnd();


                        }

void colorcube()
{
      /*  Front of MYcube counter clockwise front facing face*/
    glColor3f(1.0,0.0,0.0);       //red
    polygon(0,3,2,1);

    /*  Back of MYcube  clockwise back facing face*/
      glColor3f(0.0,1.0,0.0);      //green
      polygon(4,5,6,7);

    /*  Bottom of MYcube clockwise back facing face*/
      glColor3f(1.0,0.2,0.7);      //deep pink
      polygon(3,0,4,7);

      /*  Top of MYcube counter clockwise front facing face*/
      glColor3f(0.0,0.75388,1.0);      //blue
    polygon(1,2,6,5);

      /*  Right Side of MYcube counter clockwise front facing face*/
      glColor3f(1.0,1.0,0.0);      // yellow
    polygon(2,3,7,6);

      /*  Left Side of MYcube clockwise back facing face*/
      glColor3f(1.0,0.75,0.0);      // orange
      polygon(5,4,0,1);
}

static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
static GLdouble viewer[]= {0.0, 0.0, 5.0}; /* initial viewer location */

void display(void)
{

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* Update viewer position in modelview matrix */

      glLoadIdentity();
```

```
      gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0,
0.0);

/* rotate cube */

      glRotatef(theta[0], 1.0, 0.0, 0.0);
      glRotatef(theta[1], 0.0, 1.0, 0.0);
      glRotatef(theta[2], 0.0, 0.0, 1.0);

 colorcube();

 glFlush();
      glutSwapBuffers();
}

void mouse(int btn, int state, int x, int y)
{
      if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
      if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
      if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
      theta[axis] += 2.0;
      if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
      display();
}

void keys(unsigned char key, int x, int y)
{

/* Use x, X, y, Y, z, and Z keys to move viewer */

   if(key == 'x') viewer[0]-= 1.0;
   if(key == 'X') viewer[0]+= 1.0;
   if(key == 'y') viewer[1]-= 1.0;
   if(key == 'Y') viewer[1]+= 1.0;
   if(key == 'z') viewer[2]-= 1.0;
   if(key == 'Z') viewer[2]+= 1.0;
   display();
}

void myReshape(int w, int h)
{
 glViewport(0, 0, w, h);

/* Use a perspective view */

 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
      if(w<=h) glFrustum(-2.0, 2.0, -2.0 * (GLfloat) h/ (GLfloat) w,
       2.0* (GLfloat) h / (GLfloat) w, 2.0, 20.0);
      else glFrustum(-2.0, 2.0, -2.0 * (GLfloat) w/ (GLfloat) h,
       2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0);

/* Or we can use gluPerspective */

 /* gluPerspective(45.0, w/h, -10.0, 10.0); */

 glMatrixMode(GL_MODELVIEW);
```

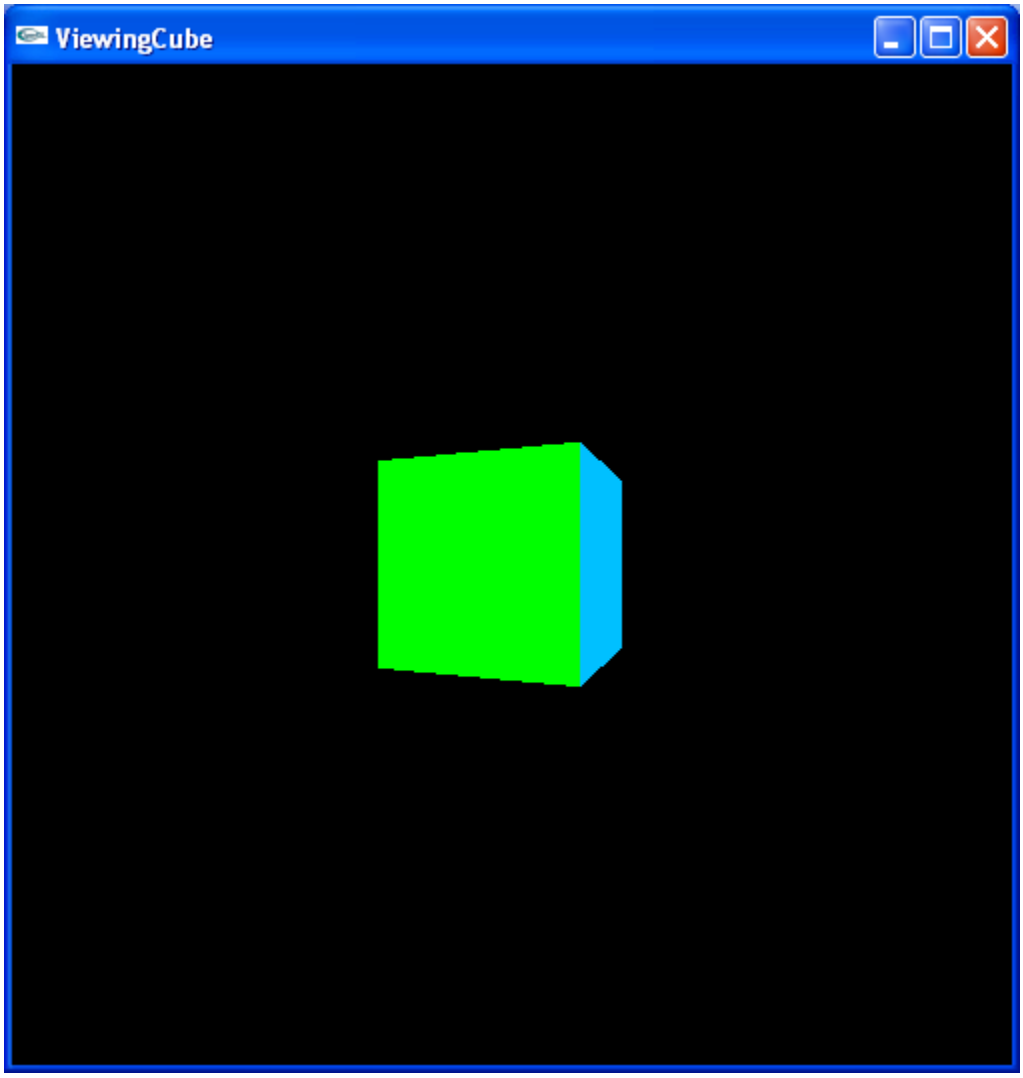
```
}

void
main(int argc, char **argv)
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
 glutInitWindowSize(500, 500);
 glutCreateWindow("ViewingCube");
      glutReshapeFunc(myReshape);
      glutDisplayFunc(display);
      glutMouseFunc(mouse);
      glutKeyboardFunc(keys);
 glEnable(GL_DEPTH_TEST);
 glutMainLoop();
}
```



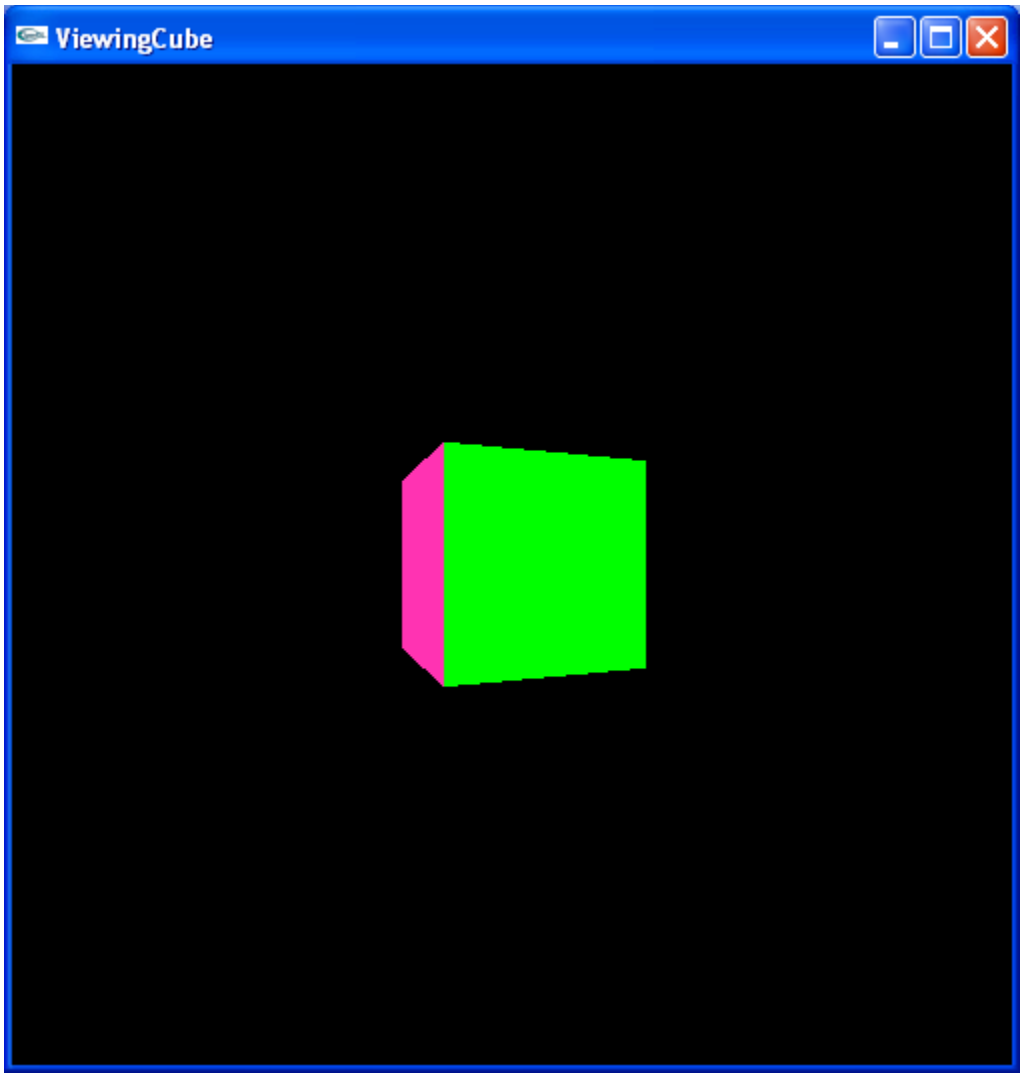
B2. With screenshots, give the gluLookAt parameters after picking each x, X, y, Y, z, Z twice.

ANSWER:

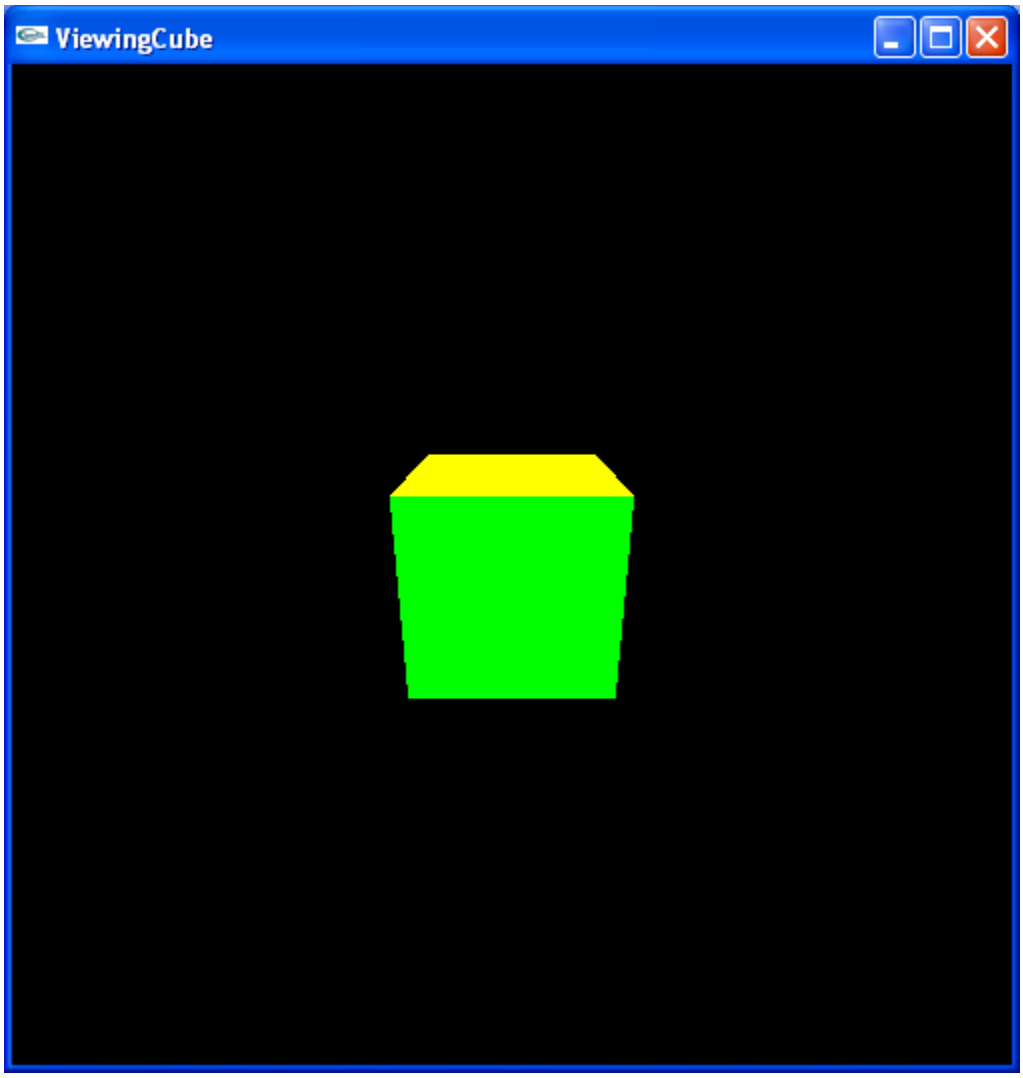
X twice:



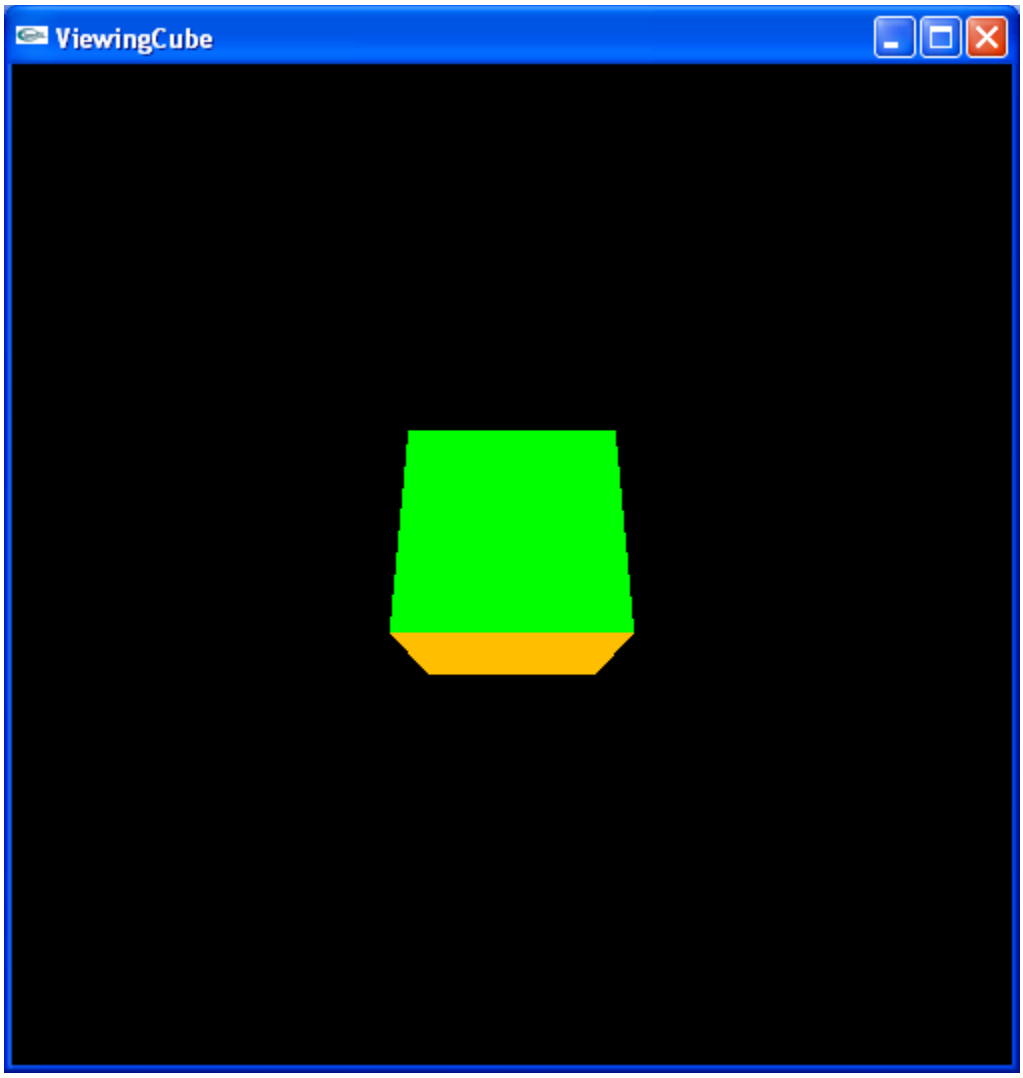
x twice:



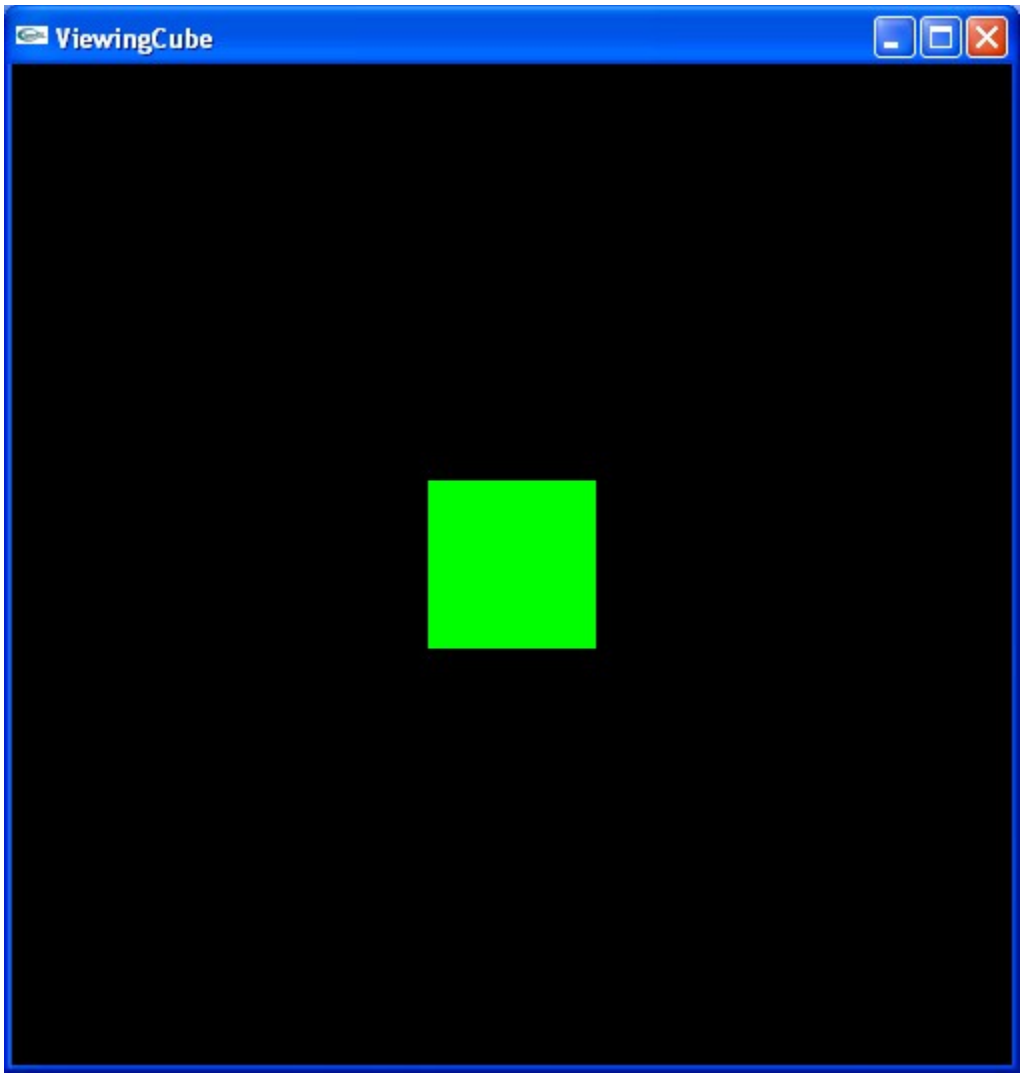
Y twice:



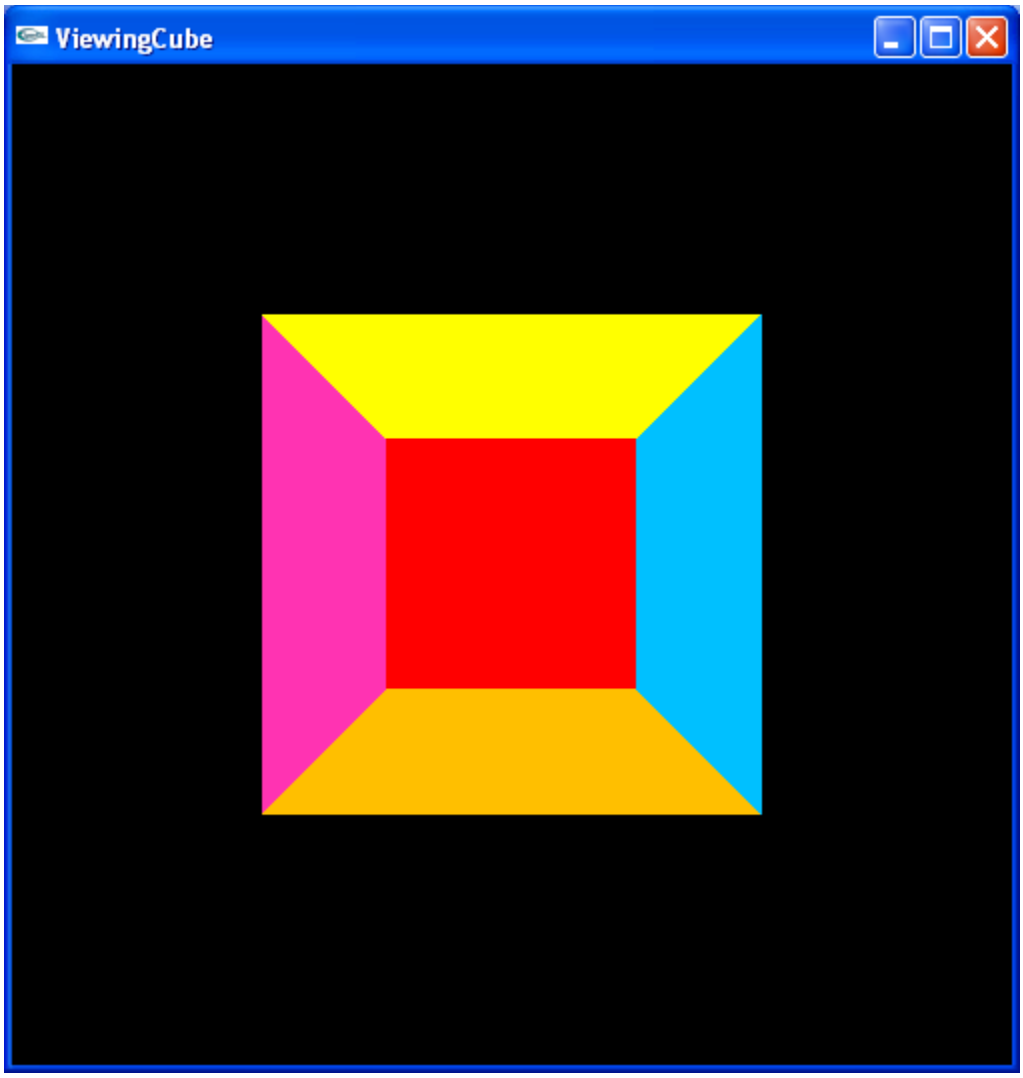
y twice:



Z twice:



z twice:


ViewingCube